D7

Fri Oct 10 14:48:14 2008 Function Index Page 2

Fri Oct 10 14:48:14 2008 Function Index Page 1

Fri Oct 10 14:48:14 2008 Function Index Page 2

```
/*****************************************************************
                                                          ******
**
** File Name:   RSLmarkunm.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**       This module contains the Restore Service Library functions to
                                                          mark and
**       unmark objects for restoral.
**
** Table of Contents:
** ------------------
**       Service Library Functions:
**               RSTSL_MarkObject
**               RSTSL_UnmarkObject
**
**       Internal Functions:
**        int    allowed_to_mark(
                        struct recover_context *rcx, tree_node *tnp,
**                              char *name_if_known,
                                                  rbcat_elem_t *clmp)
** static int    mtx(struct mcat_applyfunc_args *ctxp)
** static int    umtx(struct mcat_applyfunc_args *ctxp)
** static int    mark_tree(
                        struct recover_context *rcx, tree_node *tnp,
**                       char *name_if_known)
** static int    unmark_tree(
                        struct recover_context *rcx, tree_node *tnp)
**
**
** Compile-Time Options:
**              This section must list any compile time definitions
**              which will affect this header.
**
**
** NOTE: Part of this module is adapted from:
** server/libs/recover/grandfathered/cmd_markunmark.c
** It contains mainly support routines needed by the mark and unmark
** API functions.
**
*****************************************************************
                                                          *****/



/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */
```



```
#define  _POSIX_SOURCE 1


/*
 * System headers.
 */


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>


/*
 * Local headers
 */
#include <RSLinterns.h>
#include <RSLrbrmain.h>


/*
 * #defines, structures, typedefs local to this source file
 */

struct mark_context
{
    boolean_ty no_badfiles;       /* don't mark bad files */
    boolean_ty no_descend;
                                  /* don't descend dirs, just mark them */
    ulong_t n_this;
    ulong_t n_before;
};

static int
mark_tree(struct recover_context *rcx,
          tree_node *tnp,
          char *name_if_known);

static int
unmark_tree(struct recover_context *rcx,
            tree_node *tnp);


/*
 * External declarations
 */

/*
 * Global variables for progress of marks/unmarks
 */
static RSTSL_MarkProgressProc global_progress_cb;
static boolean_ty global_was_cancelled;
static unsigned long global_start_marks;

NEW_SRC_FILE();


/*
 * Local function prototypes
 */

static int    mtx(struct mcat_applyfunc_args *ctxp);
static int    umtx(struct mcat_applyfunc_args *ctxp);
static void MarkUnmarkDebugLogEcho(tree_node *tnp);
```

```
125 static int check_parent_perms(struct recover_context *rcx,
126                                 char *name_if_known, char *errorbuff);
```



```
128 /*********************************************
129  *
130  * allowed_to_mark:
131  *
132  *  Does the recovering user have permission to a file
133  *  for recovery.
134  *
135  * returns
136  *  1 if allowed to mark, 0 if permission is denied.
137  *
138  * Parameters:
139  *  rcx (I)           -- recover_context
140  *  tnp (I)           -- tree node pointer
141  *  name_if_known (I) -- name of file
142  *  clmp (I)          -- catalog elem
143  *********************************************/

145 int
146 allowed_to_mark(struct recover_context  *rcx,
147                 tree_node               *tnp,
148                 char                    *name_if_known,
149                 rbcat_elem_t            *clmp)
150 1 {
151 1     int ok = 1;

153 1     /*
154 1      * Modification for bug no OSGsw 23406
155 1      * perm_reason initialised to empty string
156 1      */
157 1     char perm_reason[256] = "";
158 1     /*
159 1      * Modifications for bug no OSGsw23406 end
160 1      */


163 1     if (S_ISREG(clmp->ce_mode))
164 2     {
165 2          /*
166 2           *  For regular files read permision required.
167 2           */

169 2          ok = rcx_permchk(rcx, PERMCHK_R, tnp, &clmp);
170 1     }
171 1     else if (S_ISDIR(clmp->ce_mode))
172 2     {
173 2          /*
174 2           *  For directories execute and read permision required.
175 2           */

177 2          ok = rcx_permchk(rcx, PERMCHK_R|PERMCHK_X, tnp, &clmp);
178 1     }
179 1     else if (S_ISBLK(clmp->ce_mode) || S_ISCHR(clmp->ce_mode))
180 2     {
181 2          /*
182 2           *  For char and block devices,
                                        only super-user can extract these.
183 2           */

185 2          ok = (rcx->rc_effective_uid == 0);
186 2          strcpy(perm_reason, "to recover you must be super-user");
187 1     }
188 1     else
189 2     {
190 2          /*
191 2           *  For unexpected file types no permission checks and we log
                                                        this one.
```

```
192  2          */

194  2        char pathbuf[EB_MAXPATHLEN];
195  2        char *pbp = pathbuf;


198  2        tn_getpath(tnp, &pbp);

200  2        ok = 1;
201  2        rbe_log_stats(0,
202  2                      "Unexpected File Type,
                                no permission checks to mark file %s", pbp);
203  1      }

205  1      if (!ok)
206  2      {

208  2      /*
209  2       * Modifications for bug no OSGsw23406
210  2       */
211  2           if (NULL != name_if_known)
212  3           {
213  3               rbe_log_stats(0,
214  3                             "Permission denied for file \"%s\" %s.",
215  3                             name_if_known, perm_reason);
216  2          }
217  2           else
218  3           {
219  3               char pathbuf[EB_MAXPATHLEN];
220  3               char *pbp = pathbuf;


223  3               tn_getpath(tnp, &pbp);
224  3               rbe_log_stats(0,
225  3                             "Permission denied for file \"%s\" %s.",
226  3                             pbp, perm_reason);
227  2           }
228  2      /*
229  2       * Modifications for bug no OSGsw23406 end
230  2       */

232  1      } /* end of if not ok */

234  1      return ok;
235    }         /* end of allowed_to_mark() */
```



```
237    /*******************************************
238     *
239     * mtx:
240     *
241     *
242     *******************************************/

244    static int
245    mtx(struct mcat_applyfunc_args *ctxp)
246  1 {
247  1     struct recover_context *rcx =
248  1           LINTABLE_CAST(struct recover_context *, ctxp->arg);

250  1     return mark_tree(rcx, ctxp->tnp, (char *)NULL);
251    }          /* end of mtx() */
```

```
253  /*********************************************
254   *
255   * umtx:
256   *
257   *
258   *********************************************/
259
260  static int
261  umtx(struct mcat_applyfunc_args *ctxp)
262 1 {
263 1     struct recover_context *rcx =
264 1          LINTABLE_CAST(struct recover_context *, ctxp->arg);
265
266 1     return unmark_tree(rcx, ctxp->tnp);
267  }         /* end of umtx() */
```



```
269  /*
270   * The Below are static variables global to this module api_markunmark.c
271   *  They get initialized in RSTSL_MarkObject to 0.
272   *  They get incremented in mark_tree() when the conditions of mark are
273   *  encountered.
274   *  They get assigned to the output parameters to RSTSL_MarkObject after
275   *  the mark_tree call.
276   */
277
278  static unsigned long PermissionDeniedFileCount;
279  static unsigned long BADDATAFileCount;
280
281  /*********************************************
282   *
283   * mark_tree:
284   *
285   *  Provides mark function that is based on the current mcat
286   *  recover context. The mark operation has hidden parameters
287   *  in the mark_context struct.
288   *
289   * Globals:
290   *  PermissionDeniedFileCount and BADDATAFileCount
291   *  are statics global to this module.
292   *
293   * returns 0 for success always.
294   *
295   * Parameters:
296   *  rcx (I) -- recover_context
297   *  tnp (I) -- tree node pointer
298   *  name_if_known (I) -- name of imput file.
299   *********************************************/
300
301  static int
302  mark_tree(struct recover_context *rcx,
303            tree_node *tnp,
304            char *name_if_known)
305 1 {
306 1     struct mark_context *mcp =
307 1          LINTABLE_CAST(struct mark_context *, rcx->rc_cmd_context);
308
309 1     eperrno ep_status = E_SUCCESS;
310
311
312 1   /*
313 1    * Check to see if the mark operation has been cancelled,
314 1    * If so, return without marking.
315 1    *
316 1    * This is due to the recursiveness of the mcat_apply2children
317 1    * function to stop any further processing.
318 1    */
319
320 1   if (global_was_cancelled)
321 2   {
322 2     return (0);
323 1   }
324
325 1     /*
326 1      * Only do work if element not already marked.
327 1      */
328
329 1     if (! TMARKED(rcx->rc_marks, tnp))
330 2     {
```

```
331 2           rbcat_elem_t clm;

333 2           /*
334 2            * get catalog record
335 2            */

337 2           mcat_getcatlm(rcx->rc_mcp, tnp, &clm);

339 2           /*
340 2            * Skip file if it's bad and user wants to ignore bad files
341 2            */

343 2           if (mcp->no_badfiles && (CESTAT_BADDATA & clm.ce_status))
344 3           {
345 3               char pathbuf[EB_MAXPATHLEN];
346 3               char *pbp = pathbuf;


349 3               tn_getpath(tnp, &pbp);
350 3               rbe_log_stats(
                       0, "The file %s has BADDATA and was not marked", pbp);
351 3               BADDATAFileCount++;
352 3               return 0;
353 2           }

355 2           if (CESTAT_BADDATA & clm.ce_status)
356 3           {
357 3               char pathbuf[EB_MAXPATHLEN];
358 3               char *pbp = pathbuf;


361 3               tn_getpath(tnp, &pbp);
362 3               rbe_log_stats(
                           0, "The file %s is BADDATA and was marked", pbp);
363 3               BADDATAFileCount++;
364 2           }

366 2           /*
367 2            * Check permissions.  Try to avoid calling permission
368 2            * functions by making some trivial checks in line
369 2            * (owner can always mark; root can always mark).
370 2            */

372 2           if (rcx->rc_effective_uid != (uid_t)clm.ce_owner
373 2               && rcx->rc_effective_uid != 0
374 2               && ! allowed_to_mark(rcx, tnp, name_if_known, &clm))
375 3           {
376 3               /*
377 3                * logging done in allow_to_mark
378 3                */

380 3               PermissionDeniedFileCount++;
381 3               return 0;
382 2           }

384 2           /*
385 2            * 1) Setting mark this bitfile for recovery bit.
386 2            * 2) incrementing marks for the plane
387 2            * 3) incrementing total marks
388 2            * 4) incrementing the operated on bitfile count for call to
                                                                   mart_tree
389 2            */

391 2           TSETMARK(rcx->rc_marks, tnp);
392 2           rcx->rc_nmarks_by_plane[-tnp->tn_mcplane]++;
393 2           rcx->rc_nmarks_total++;
```



```
394 2           mcp->n_this++;

396 2           /*
397 2            * To avoid double logging on mark, do not log BADDATA files.
398 2            */

400 2           if (! CESTAT_BADDATA & clm.ce_status)
401 3           {
402 3                MarkUnmarkDebugLogEcho(tnp);
403 2           }

405 2           /*
406 2            * If this is a DS_NONE record and has no catlm,
407 2            * then we need to remember where we can find the
408 2            * actual catlm information for this node.
409 2            */

411 2           if (tnp->tn_tlm_catlm == -1)
412 3           {
413 3               eerrno_ty ret_status;


416 3               if (E_SUCCESS != (ret_status = add_dsnone(rcx, tnp)))
417 4               {
418 4                  return ret_status;
419 3               }
420 2           }

422 2           /*
423 2            * If were marking lets add the bfile to the volid list.
424 2            * if needed adding the volume to the list and incrementing
425 2            * the bfile dependacy count.
426 2            */

428 2           rcx->ebvllist = ebvl_add_bfile_to_volid_count (
                                                          rcx-> ebvllist,
429 2                                        (ebfs_uid_ty*) &(
                                                  clm.ce_bitfileID),
430 2                                        &ep_status);
431 2           if (E_SUCCESS != ep_status)
432 3           {
433 3               /*
434 3                * This error does not effect the recover,
                                                          but lets log it
435 3                */

437 3               rbe_internal_error( ep_status,
438 3                   "NOTE: in ebvl_add_bfile_to_volid_count(
                                                          ) for bitfile %s",
439 3                   eb_ebfsid2ascii(&clm.ce_bitfileID, NULL,
440 3                                   EB_EBFSID2ASCII_WITH_DOTS) );
441 2           }

443 2           /*
444 2            * if the summary is valid, update it for this bfile.
445 2            */

447 2           if (rcx->rc_mark_summary_valid)
448 3           {
449 3               add_to_summary(&rcx->rc_mark_summary, &clm);
450 2           }

452 2           /*
453 2            * If we have reached the boundary for reporting progress
454 2            * prior to marking this file, call the progress routine
455 2            * and verify it is OK to continue
```

```
456 2            */
457 2           if (((
                    rcx->rc_nmarks_total - global_start_marks) & 0x3ff) == 0)
458 3           {
459 3             /* Don't bother on 0 */
460 3             if ((rcx->rc_nmarks_total - global_start_marks) > 0)
461 4             {
462 4               /* If callback returns non-zero,
                                        flag this operation as cancelled */
463 4               if (global_progress_cb (
                              rcx->rc_nmarks_total - global_start_marks))
464 5               {
465 5                 /* Set the cancelled flag */
466 5                 global_was_cancelled = TRUE;

468 5                 /* Get out, since user cancelled */
469 5                 return (0);
470 4               }
471 3             }
472 2           }

474 1       } /* end of if (!TMARKED()) */

476 1       if (tnp->tn_flags & TNF_KNOWN_NOCHILDREN)
477 2       {
478 2           return 0;
479 1       }

481 1      /*
482 1       * Should we descend to chidren.
483 1       */

485 1       if (!mcp->no_descend)
486 2       {
487 2           (void)mcat_apply2children(rcx->rc_mcp, tnp, mtx, (char *)rcx);
488 1       }
489 1       return 0;
490     }       /* end of mark_tree() */
```



```
493     /*
494      * The Below are static variables global to this module api_markunmark.
                                                                          c
495      *  They get initialized in EDMRST_UnmarkObject to 0.
496      *  They get incremented in unmark_tree(
                                                  ) when the badfiles are encountered.
497      *  They get assigned to the output parameters to EDMRST_UnmarkObject
                                                                        after
498      *  the unmark_tree call.
499      */

501     static long BADDATAFileUnmarkCount;

503     /*********************************************
504      *
505      * unmark_tree:
506      *
507      *  Provides unmark function that is based on the current mcat
508      *  recover context. The unmark operation has hidden parameters
509      *  in the mark_context struct.
510      *
511      * Globals:
512      *  BADDATAFileUnmarkCountis a static global to this module.
513      *
514      * returns 0 for success and -1 if there are no marks to unmark.
515      *
516      * Parameters:
517      *  rcx (I) -- recover_context
518      *  tnp (I) -- tree node pointer
519      *
520      *********************************************/

522     static int
523     unmark_tree(struct recover_context *rcx,
524                 tree_node *tnp)
525 1   {
526 1       struct mark_context *mcp =
527 1            LINTABLE_CAST(struct mark_context *, rcx->rc_cmd_context);

529 1       eperrno ep_status = E_SUCCESS;


532 1      /*
533 1       * Check to see if the unmark operation has been cancelled,
534 1       * If so, return without unmarking.
535 1       *
536 1       * This is due to the recursiveness of the mcat_apply2children
537 1       * function to stop any further processing.
538 1       */

540 1      if (global_was_cancelled)
541 2      {
542 2        return (0);
543 1      }

545 1        /*
546 1         * If there are no more things marked,
                                                    don't need to go any further
547 1         */

549 1        if (rcx->rc_nmarks_total == 0)
550 2        {
551 2            return -1;
552 1        }
```

```
555  1      /*
556  1       * unmark this guy
557  1       */

559  1      if (TMARKED(rcx->rc_marks, tnp))
560  2      {
561  2           rbcat_elem_t clm;

563  2           /*
564  2            * get catalog record
565  2            */

567  2           mcat_getcatlm(rcx->rc_mcp, tnp, &clm);

569  2           /*
570  2            * Skip file if we're only unmarking bad files
571  2            * and this one is NOT bad.
572  2            */

574  2           if (mcp->no_badfiles && !(CESTAT_BADDATA & clm.ce_status))
575  3           {
576  3               goto unmark_children;
577  2           }

579  2           /*
580  2            * Count bad files
581  2            */

583  2           if (CESTAT_BADDATA & clm.ce_status)
584  3           {
585  3               BADDATAFileUnmarkCount++;

587  3               /*
588  3                * Log if this is an unmark BADDATA files only unmark
                                                                        call
589  3                *
590  3                */

592  3               if (mcp->no_badfiles)
593  4               {
594  4                   char pathbuf[EB_MAXPATHLEN];
595  4                   char *pbp = pathbuf;


598  4                   tn_getpath(tnp, &pbp);
599  4                   rbe_log_stats(
                                0, "The BADDATA file %s was umarked", pbp);
600  3               }
601  2           }

603  2           /*
604  2            * 1) Clearing mark this bitfile for recovery bit.
605  2            * 2) decrementing marks for the plane
606  2            * 3) decrementing total marks
607  2            * 4) incrementing the unmarked bitfile count for call to
                                                                  unmark_tree
608  2            */

610  2           TCLRMARK(rcx->rc_marks, tnp);
611  2           rcx->rc_nmarks_by_plane[-tnp->tn_mcplane]--;
612  2           rcx->rc_nmarks_total--;
613  2           mcp->n_this++;

615  2           /*
```



```
616  2           * To avoid double logging on mark, do not log BADDATA files.
617  2           * if BadfileOnly has been set.
618  2           */

620  2          if (!(mcp->no_badfiles) && (CESTAT_BADDATA & clm.ce_status))
621  3          {
622  3              MarkUnmarkDebugLogEcho(tnp);
623  2          }

625  2          /*
626  2           * If were unmarking lets decrement the bfile to the volid
                                                                        list
627  2           * bfile dependacy count.
628  2           */

630  2          ep_status = ebvl_decr_bfile_to_volid_count (rcx-> ebvllist,
631  2                                                      (ebfs_uid_ty*) &(
                                                         clm.ce_bitfileID));
632  2          if (E_SUCCESS != ep_status)
633  3          {
634  3              /*
635  3               * This error does not effect the recover,
                                                          but lets log it
636  3               */

638  3              rbe_internal_error( ep_status,
639  3                  "NOTE: in ebvl_decr_bfile_to_volid_count(
                                                          ) for bitfile %s",
640  3                  eb_ebfsid2ascii(&clm.ce_bitfileID, NULL,
641  3                                  EB_EBFSID2ASCII_WITH_DOTS) );
642  2          }

644  2          /*
645  2           * if the summary is valid, update it for this bfile.
646  2           */

648  2          if (rcx->rc_mark_summary_valid)
649  3          {
650  3              sub_from_summary(&rcx->rc_mark_summary, &clm);
651  2          }

653  2          /*
654  2           * undo dsnone info saved, if any
655  2           */

657  2          if (tnp->tn_tlm_catlm == -1)
658  3          {
659  3              remove_dsnone(rcx, tnp);
660  2          }

662  2          /*
663  2           * If we have reached the boundary for reporting progress
664  2           * prior to unmarking this file, call the progress routine
665  2           * and verify it is OK to continue
666  2           */

668  2          if (((
                   global_start_marks - rcx->rc_nmarks_total) & 0x3ff) == 0)
669  3          {
670  3            /* Don't bother on 0 */
671  3            if ((global_start_marks - rcx->rc_nmarks_total) > 0)
672  4            {
673  4              /* If callback returns non-zero,
                                          flag this operation as cancelled */
674  4              if (global_progress_cb (
                                global_start_marks - rcx->rc_nmarks_total))
```

```
675  5              {
676  5                /* Set the cancelled flag */
677  5                global_was_cancelled = TRUE;

679  5                /* Get out, since user cancelled */
680  5                return (0);
681  4              }
682  3            }
683  2          }
684  1      }

686  1  unmark_children:

688  1      if (tnp->tn_flags & TNF_KNOWN_NOCHILDREN)
689  2      {
690  2          return 0;
691  1      }

693  1      if (!mcp->no_descend)
694  2      {
695  2          (void)mcat_apply2children(rcx->rc_mcp, tnp, umtx, (
                                                              char *)rcx);
696  1      }
697  1      return 0;
698    }   /* end of unmark_tree() */
```



```
700    static int
701    check_parent_perms(struct recover_context *rcx,
702                       char *name_if_known,
703                       char *errorbuff)
704  1 {
705  1     rbcat_elem_t clm;
706  1     rbcat_elem_t *clmp = &clm;
707  1     tree_node *tnp;
708  1     char      parent_path[PATH_MAX], *tmpptr;
709  1     int       ok = 1;


712  1     if (rcx->rc_effective_uid == 0)
713  2     {
714  2         return 0;
715  1     }

717  1     strcpy(parent_path, name_if_known);

719  1     if ((tmpptr = strrchr(parent_path, '/')) == NULL)
720  2     {
721  2          return 0;
722  1     }

724  1     do
725  2     {
726  2         if (tmpptr == parent_path)
727  3          {
728  3             return 0;
729  2          }

731  2         *tmpptr = 0;

733  2         tnp = mcat_lookup_path(rcx->rc_mcp, parent_path);

735  2         /*
736  2          * get catalog record
737  2          */

739  2         if (tnp != NULL)
740  3         {
741  3             mcat_getcatlm(rcx->rc_mcp, tnp, &clm);

743  3             if (rcx->rc_effective_uid != clm.ce_owner)
744  4              {
745  4                 ok = rcx_permchk(rcx, PERMCHK_R|PERMCHK_X, tnp, &clmp);
746  3              }
747  2         }
748  1     }
749  1     while (ok && (tmpptr = strrchr(parent_path, '/')) != NULL);

751  1     if (ok == 0 && errorbuff != NULL)
752  2     {
753  2         strcpy(errorbuff, parent_path);
754  1     }

756  1     return !ok;
757    }        /* check_parent_perms */
```

```
760  /*****************************************************************
761   * RSTSL_MarkObject()
762   *
763   *   The MarkObject operation takes a restorableObject and marks it, and
764   *   possibly its descendant files for restoral based on the input criteria.
765   *   The RSTSL_MarkObject call is an asynchronously executed operation
766   *   in the Restore Engine that performs the marking. It stores its results
767   *   in buffers supplied by the caller, so that the synchronous RPC thread can
768   *   test for and retrieve results without another Restore Service Library
769   *   call.
770   *
771   * Parameters:
772   *
773   *  thisObject    (I) - The restoral object; can be a leaf object (e.g. a
774   *                      file), or a container object (e.g., a directory).
775   *  time          (I) - (optional) the backup time to perform the mark on --
776   *                      if not specified, uses currrently selected backup; if
777   *                      specified, leaves selected backup time unchanged
778   *  allowBadfiles (I) - allows marking of files of state BADDATA.
779   *  descend       (I) - Should mark operation descend to operate on the content
780   *                      of container objects.
781   *  BadFilesCount (O) - returns the file count with BADDATA.
782   *  PermDenyFilesCount (O) -- returns the file count with permission denied.
783   *  fileMarked    (O) - return the total files marked after this mark occurred.
784   *  dirMarked     (O) - return the total directories marked after this mark
785   *                      occurred.
786   *  otherMarked   (O) - return the total "other" files marked after this mark.
787   *  progressCB    (I) - pointer to callback function to report progress and
788   *                      test for cancellation
789   *
790   *****************************************************************/

792  eerrno_ty
793  RSTSL_MarkObject( struct RSTRPC_user_restorable_object *thisObject,
794                    time_t          backupTime,
795                    boolean_ty      allowBadFiles,
796                    boolean_ty      descend,
797                    unsigned long *BadFilesCount,
798                    unsigned long *PermDenyFilesCount,
799                    unsigned long *fileMarked,
800                    unsigned long *dirMarked,
801                    unsigned long *otherMarked,
802                    RSTSL_MarkProgressProc progressCB )
803 1 {
804 1     struct mark_context mc;
805 1     boolean_ty ssdb_opened = FALSE;
806 1     char   errorbuff[PATH_MAX];
807 1     int rtc;     /* return code from some internal routines */
```



```
808 1     time_t        save_time = 0;
809 1     u_hyper       lenMarkedFiles;
810 1     RSTRPC_backup_flags_ty        backup_flags = 0;

812 1     /*
813 1      *  Output parameters check pointers against NULL & intialized to zero
814 1      */

816 1     if (!fileMarked || !dirMarked || !otherMarked)
817 2     {
818 2          return EP_RB_RECOVER_BAD_ARGS;
819 1     }

821 1     *fileMarked = *dirMarked = *otherMarked=0;

823 1     if (!BadFilesCount || !PermDenyFilesCount)
824 2     {
825 2          return EP_RB_RECOVER_BAD_ARGS;
826 1     }

828 1     /*
829 1      * lets validate the restorableObject as an input parameter.
830 1      */

832 1      if (NULL == thisObject)
833 2      {
834 2          return EP_RB_RECOVER_BAD_ARGS;
835 1      }

837 1      if (thisObject->root.objName == NULL)
838 2      {
839 2           return (EP_RB_RECOVER_INVAL_OBJNAME);
840 1      }

842 1      switch (thisObject->root.objLevel)
843 2      {
844 2      case RSTRPC_container_type:
845 2      case RSTRPC_leaf_type:
846 2          /* make sure input object is from current TLO's backup app */
847 2           if (rcp->rc_backup_app != thisObject->root.backupApp)
848 2                 return (EP_RB_RECOVER_INVALOP);
849 2           break;

851 2      case RSTRPC_tlo_type:
852 2           return (EP_RB_RECOVER_INVALOP);

854 2      default:
855 2           return (EP_RB_RECOVER_INVAL_OBJTYPE);
856 1      }

858 1     /* check if backup time must change */
859 1     if ( backupTime != (time_t)0 )
860 2     {
861 2          /* get current backup time; if different from requested time,
862 2           * then set requested time and save current time; at end, restore
863 2           * current time */

865 2          if (rcp->rc_backup_app != 0)
866 3          {
867 3                  rtc = rcp-> currentPIptr-> piFuncArray[PIFuncIndexGetCurTime ]
868 3                                                        (rcp, &save_time);
869 3                  if (rtc == E_SUCCESS)
```

```
870 3                           rtc = rcp-> currentPIptr->
871 3                                   piFuncArray[
                                             PIFuncIndexSetBkupTime ]
872 3                                      (
                                        rcp, backupTime, backup_flags);
873 3           } else {
874 3                   rtc = RSTSL_GetCurrentBackupTime( &save_time );
875 3                   if (rtc == E_SUCCESS)
876 3                           rtc = RSTSL_SetBackupForTime (
                                        backupTime, backup_flags);
877 2           }
878 2           if (rtc != E_SUCCESS)
879 2                   return rtc;
880 1       }

882 1       /* if this isn't a network backup restore,
                                        call plugin to do mark */
883 1       if (rcp->rc_backup_app != 0)
884 2       {
885 2            rtc = rcp-> currentPIptr-> piFuncArray[ PIFuncIndexMark ]
886 2                          (
                             rcp, thisObject, allowBadFiles, descend,
887 2                           BadFilesCount, PermDenyFilesCount,
                                                    fileMarked,
888 2                          &lenMarkedFiles, dirMarked,
                                                   otherMarked,
889 2                          progressCB );

891 2            /* save mark summary in restore_context */
892 2            rcp->rc_mark_summary.nfiles = *fileMarked ;
893 2            rcp->rc_mark_summary.ndirs = *dirMarked;
894 2            rcp->rc_mark_summary.nother = *otherMarked;
895 2            rcp->rc_mark_summary.len_mkd_files.high = lenMarkedFiles.high;
896 2            rcp->rc_mark_summary.len_mkd_files.low = lenMarkedFiles.low;

898 2            if (save_time)
899 2                    rcp-> currentPIptr-> piFuncArray[
                                        PIFuncIndexSetBkupTime ]
900 2                                      (
                                    rcp, save_time, backup_flags);

902 2            return rtc;
903 1       }


906 1       /* Initial global marking/unmarking cancel flag to FALSE */
907 1       global_was_cancelled = FALSE;
908 1       global_progress_cb = progressCB;
909 1       global_start_marks = rcp->rc_nmarks_total;


912 1       mc.n_this = 0;
913 1       mc.n_before = rcp->rc_nmarks_total;

915 1       *BadFilesCount = *PermDenyFilesCount = 0;

917 1      /*
918 1       * The Below are static variables global to this module
                                                    api_markunmark.c
919 1       *  They get initialized in EDMRST_MarkObject to 0.
920 1       *  They get incremented in mark_tree(
                                        ) when the conditions of mark are
921 1       *  encountered.
922 1       *  They get assigned to the output parameters to EDMRST_MarkObject
                                                              after
923 1       *  the mark_tree call.
```



```
924 1       */

926 1       BADDATAFileCount = PermissionDeniedFileCount = 0;

928 1       rcp->rc_cmd_context = (char *)&mc;

930 1       /*
931 1        * input parameters
932 1        * 1) allowBadfiles --  does marking of Badfiles
933 1        * 2) descend --  does mark descend into the contents of
                                                              directories.
934 1        */

936 1       if (!allowBadFiles)
937 2       {
938 2            mc.no_badfiles = TRUE;
939 1       }
940 1       else
941 2       {
942 2            mc.no_badfiles = FALSE;
943 1       }

945 1       if (!descend)
946 2       {
947 2            mc.no_descend = TRUE;
948 1       }
949 1       else
950 2       {
951 2            mc.no_descend = FALSE;
952 1       }

954 1       /*
955 1        * Open up the saveset db during mark command.
956 1        */

958 1       rtc = ss_open_saveset_db();
959 1       if (0 != rtc)        /* ss_open_saveset_db failed */
960 2       {
961 2            return (EP_RB_RECOVER_CANT_OPEN_SSDB);
962 1       }

964 1       /*
965 1        * ssdb_opened was used to indicate whether or not
966 1        * ss_open_saveset_db was successful. It was set to TRUE
967 1        * when ss_open_saveset_db() succeeded, so we'll remember
968 1        * to call ss_close_saveset_db later. This is no longer
969 1        * necessary because we would have returned an error if
970 1        * ss_open_saveset_db failed. It's left in to minimize
971 1        * code changes.
972 1        */

974 1       ssdb_opened = TRUE;

976 1       /*
977 1        * validate restorable object with the current mcat context
978 1        */

980 2       {
981 2            /*
982 2             * Initialize tnp_thisObject to NULL so that if
                                                        mcat_lookup_path
983 2             * does not set it to NULL in the case of failure, the code
984 2             * can work correctly.
985 2             */
986 2            tree_node *tnp_thisObject = NULL;
```

```
988  2          tnp_thisObject = mcat_lookup_path(
                                    rcp->rc_mcp, thisObject->root.objName);
989  2          if (tnp_thisObject == NULL)
990  3          {
991  3              /*
992  3               *   If we opened ssdb then lets close it!
993  3               *
994  3               *   restorable object must be corrupt mcat_lookup_path
                                                                  could not
995  3               *   locate the tree node ptr for files
996  3               */
997  3              if (ssdb_opened)
998  4              {
999  4                  ss_close_saveset_db();
1000 3              }
1001 3              return EP_RB_RECOVER_BAD_CONTEXT;
1002 2          }

1004 2          if (tnp_thisObject->tn_mcplane !=
1005 2              ((netBackupObjData *)(
                            thisObject->appData.data))->objtnMCplane)
1006 3          {
1007 3              /*
1008 3               * If we opened ssdb then lets close it!
1009 3               *
1010 3               * restorable object must be out of context, return error
1011 3               */

1013 3              if (ssdb_opened)
1014 4              {
1015 4                  ss_close_saveset_db();
1016 3              }
1017 3              return EP_RB_RECOVER_BAD_CONTEXT;
1018 2          }

1020 2          /*
1021 2           * If we made it here then thisObject is valid.
1022 2           */

1024 2          if (check_parent_perms(
                              rcp, thisObject->root.objName, errorbuff))
1025 3          {
1026 3              rbe_log_stats(0,
1027 3                            "Permission denied for file \"%s\" %s.",
1028 3                            thisObject->root.objName,
1029 3                            "parent directory permissions");
1030 2          }
1031 2          else
1032 3          {
1033 3              /*
1034 3               * mark_tree always returns 0
1035 3               */

1037 3              (void) mark_tree(
                          rcp, tnp_thisObject, thisObject->root.objName);
1038 2          }
1039 1      }

1041 1      /*
1042 1       * The Below are static variables global to this module
                                                        api_markunmark.c
1043 1       *  They get initialized in EDMRST_MarkObject to 0.
1044 1       *  They get incremented in mark_tree(
                                        ) when the conditions of mark are
1045 1       *  encountered.
1046 1       *  They get assigned to the output parameters to
```



```
                                                    EDMRST_MarkObject after
1047 1       *  the mark_tree call.
1048 1       */

1050 1      *BadFilesCount = BADDATAFileCount;
1051 1      *PermDenyFilesCount = PermissionDeniedFileCount;

1053 1      /*
1054 1       * if the summary is valid, set the total number of marked files,
1055 1       * directories, and "other" files. This is not intended to be the
1056 1       * number of files that resulted directly from the above mark_tree
                                                                    call.
1057 1       */

1059 1      if (rcp->rc_mark_summary_valid)
1060 2      {
1061 2          *fileMarked = rcp->rc_mark_summary.nfiles;
1062 2          *dirMarked = rcp->rc_mark_summary.ndirs;
1063 2          *otherMarked = rcp->rc_mark_summary.nother;
1064 1      }
1065 1      else
1066 2      {
1067 2          /*
1068 2           * This error is not fatal, the output variable *Marked,
1069 2           * will be zero, we should never get this error.
1070 2           */

1072 2          rbe_log_stats(
    0, "Internal error: mark summary not Valid in EDMRST_MarkObject()");
1073 1      }

1075 1      /*
1076 1       * If we opened it then lets close it!
1077 1       */

1079 1      if (ssdb_opened)
1080 2      {
1081 2          ss_close_saveset_db();
1082 1      }

1084 1      if (save_time)
1085 1          RSTSL_SetBackupForTime( save_time, backup_flags );

1087 1      return( E_SUCCESS );
1088   }          /* end of RSTSL_MarkObject () */
```

```
1090  /*****************************************************************
1091   * UnmarkObject
1092   *
1093   *   The UnmarkObject operation takes a restorableObject and unmarks
                                                                    it, and
1094   *   possibly its descendant files for restoral based on the input
                                                                  criteria.
1095   *   The RSTSL_UnmarkObject call is an asynchronously executed
                                                                  operation
1096   *   in the Restore Engine that performs the unmarking.
                                                            It returns unmark
1097   *   progress and tests for user cancelation through a callback
                                                                  function.
1098   *
1099   * UnmarkObject Parameters:
1100   *
1101   *  thisObject    (I) - The restoral object;
                                          can be a leaf object (e.g. a
1102   *                     file), or a container object (
                                                  e.g., a directory).
1103   *  backupTime    (I) - (
                    optional) the backup time to perform the unmark on --
1104   *                     if not specified,
                                    uses currrently selected backup; if
1105   *                     specified,
                                  leaves selected backup time unchanged
1106   *  BadFilesOnly  (
                   I) - allows unmarking ONLY of files of state BADDATA.
1107   *  descend       (
                  I) - Should unmark operation descend to operate on the
1108   *                     content of container objects.
1109   *  BadFilesCount (O) - returns the file count with BADDATA.
1110   *  fileMarked    (
                O) - return the total files marked after this mark occurred.
1111   *  dirMarked     (
                O) - return the total directories marked after this mark
1112   *                     occurred.
1113   *  otherMarked   (
                O) - return the total "other" files marked after this mark.
1114   *  progressCB    (
                I) - pointer to callback function to report progress and
1115   *                     test for cancellation
1116   *****************************************************************/

1118  eerrno_ty RSTSL_UnmarkObject(
                        struct RSTRPC_user_restorable_object *thisObject,
1119                            const time_t          backupTime,
1120                             const boolean_ty     BadFilesOnly,
1121                             const boolean_ty     descend,
1122                            ulong                 *BadFilesCount,
1123                            ulong                 *fileMarked,
1124                            ulong                 *dirMarked,
1125                            ulong                 *otherMarked,
1126                             RSTSL_MarkProgressProc progressCB )
1127 1 {
1128 1     struct mark_context mc;
1129 1     eerrno_ty     rtc;
1130 1     time_t        save_time = 0;
1131 1     u_hyper       lenMarkedFiles;
1132 1     RSTRPC_backup_flags_ty       backup_flags = 0;

1134 1     /* Initial global marking/unmarking cancel flag to FALSE */
1135 1     global_was_cancelled = FALSE;
1136 1     global_progress_cb = progressCB;
1137 1     global_start_marks = rcp->rc_nmarks_total;
```



```
1139 1     mc.n_this = 0;
1140 1     mc.n_before = rcp->rc_nmarks_total;

1142 1     rcp->rc_cmd_context = (char *)&mc;


1145 1    /*
1146 1     *  Output parameters get intialized to zero
1147 1     */

1149 1    /*
1150 1     *  Output parameters check pointers against NULL & intialized to
                                                                      zero
1151 1     */

1153 1     if (!fileMarked || !dirMarked || !otherMarked || !BadFilesCount)
1154 2     {
1155 2          return EP_RB_RECOVER_BAD_ARGS;
1156 1     }

1158 1     /*
1159 1      * lets validate the restorable object as an input parameter.
1160 1      */

1162 1     if (NULL == thisObject)
1163 2     {
1164 2         return EP_RB_RECOVER_BAD_ARGS;
1165 1     }

1167 1     if (thisObject->root.objName == NULL)
1168 2     {
1169 2          return(EP_RB_RECOVER_INVAL_OBJNAME);
1170 1     }

1172 1     switch (thisObject->root.objLevel)
1173 2     {
1174 2     case RSTRPC_container_type:
1175 2     case RSTRPC_leaf_type:
1176 2          /* make sure input object is from current TLO's backup app */
1177 2           if (rcp->rc_backup_app != thisObject->root.backupApp)
1178 2                  return (EP_RB_RECOVER_INVALOP);
1179 2          break;

1181 2     case RSTRPC_tlo_type:
1182 2          return (EP_RB_RECOVER_INVALOP);

1184 2     default:
1185 2          return (EP_RB_RECOVER_INVAL_OBJTYPE);
1186 1     }

1188 1     /* check if backup time must change */
1189 1     if ( backupTime != (time_t)0 )
1190 2     {
1191 2          /* get current backup time; if different from requested time,
1192 2           * then set requested time and save current time;
                                                                  at end, restore
1193 2           * current time */

1195 2          if (rcp->rc_backup_app != 0)
1196 3          {
1197 3                  rtc = rcp-> currentPIptr-> piFuncArray[
                                                      PIFuncIndexGetCurTime ]
1198 3                                                          (
                                                    rcp, &save_time);
1199 3                  if (rtc == E_SUCCESS)
```

```
1200 3                          rtc = rcp-> currentPIptr->
1201 3                                  piFuncArray[
                                            PIFuncIndexSetBkupTime ]
1202 3                                      (
                                        rcp, backupTime, backup_flags);
1203 3          } else {
1204 3                  rtc = RSTSL_GetCurrentBackupTime( &save_time );
1205 3                  if (rtc == E_SUCCESS)
1206 3                          rtc = RSTSL_SetBackupForTime (
                                        backupTime, backup_flags);
1207 2          }
1208 2          if (rtc != E_SUCCESS)
1209 2                  return rtc;
1210 1      }

1212 1      /* if this isn't a network backup restore,
                                        call plugin to do mark */
1213 1      if (rcp->rc_backup_app != 0)
1214 2      {
1215 2          rtc = rcp-> currentPIptr-> piFuncArray[ PIFuncIndexUnmark ]
1216 2                              (
                                rcp, thisObject, BadFilesOnly, descend,
1217 2                                  BadFilesCount, fileMarked,
1218 2                                  &lenMarkedFiles, dirMarked,
                                                    otherMarked,
1219 2                                  progressCB );

1221 2          /* save mark summary in restore_context */
1222 2          rcp->rc_mark_summary.nfiles = *fileMarked ;
1223 2          rcp->rc_mark_summary.ndirs = *dirMarked;
1224 2          rcp->rc_mark_summary.nother = *otherMarked;
1225 2          rcp->rc_mark_summary.len_mkd_files.high = lenMarkedFiles.high;
1226 2          rcp->rc_mark_summary.len_mkd_files.low = lenMarkedFiles.low;

1228 2          if (save_time)
1229 2                  rcp-> currentPIptr-> piFuncArray[
                                        PIFuncIndexSetBkupTime ]
1230 2                                      (
                                    rcp, save_time, backup_flags);


1232 2          return rtc;
1233 1      }

1235 1      *fileMarked = *dirMarked = *otherMarked = *BadFilesCount = 0;

1237 1      /*
1238 1       * The Below is static variable global to this module
                                                    api_markunmark.c
1239 1       *  They get initialized in EDMRST_UnmarkObject to 0.
1240 1       *  They get incremented in unmark_tree(
                                    ) when the badfiles are encountered.
1241 1       *  They get assigned to the output parameter to
                                        EDMRST_UnmarkObject after
1242 1       *  the unmark_tree call.
1243 1       */

1245 1      BADDATAFileUnmarkCount = 0;

1247 1      /*
1248 1       * input parameters
1249 1       * 1) BadfilesOnly --  is unmark limitted to Badfiles only
1250 1       * 2) descend --  does unmark descend into the contents of
                                                        directories.
1251 1       */

1253 1      if (BadFilesOnly)
```



```
1254 2      {
1255 2          mc.no_badfiles = TRUE;
1256 1      }
1257 1      else
1258 2      {
1259 2          mc.no_badfiles = FALSE;
1260 1      }

1262 1      if (!descend)
1263 2      {
1264 2          mc.no_descend = TRUE;
1265 1      }
1266 1      else
1267 2      {
1268 2          mc.no_descend = FALSE;;
1269 1      }

1271 1      /*
1272 1       * validate restorable object with the current mcat context
1273 1       */

1275 2      {
1276 2          tree_node *tnp_thisObject;
1277 2          int ret_unmark_tree;

1279 2          tnp_thisObject = mcat_lookup_path( rcp->rc_mcp,
1280 2                                          thisObject->root.objName );
1281 2          if (tnp_thisObject == NULL)
1282 3          {
1283 3              /*
1284 3               *   restorable object must be corrupt mcat_lookup_path
                                                        could not
1285 3               *   locate the tree node ptr for files
1286 3               */

1288 3              return EP_RB_RECOVER_BAD_CONTEXT;
1289 2          }

1291 2          if (tnp_thisObject->tn_mcplane !=
1292 2              ((netBackupObjData *)(
                                thisObject->appData.data))->objtnMCplane)
1293 3          {
1294 3              /*
1295 3               * restorable object must be out of context return error
1296 3               */

1298 3              return EP_RB_RECOVER_BAD_CONTEXT;
1299 2          }

1301 2          /*
1302 2           * If we made it here then thisObject is valid.
1303 2           * unmark_tree always returns 0
1304 2           */

1306 2          (void) unmark_tree(rcp, tnp_thisObject);
1307 1      }

1309 1      /*
1310 1       * The Below are static variable global to this module
                                                    api_markunmark.c
1311 1       *  They get initialized in EDMRST_UnmarkObject to 0.
1312 1       *  They get incremented in unmark_tree(
                                        ) when the badfiles are encountered.
1313 1       *  They get assigned to the output parameters to
                                                EDMRST_UnmarkObject after
1314 1       *  the unmark_tree call.
```

```
1315 1       */

1317 1      *BadFilesCount=BADDATAFileUnmarkCount;

1319 1      /*
1320 1       * if the summary is valid, set the total number of marked files,
1321 1       * directories, and "other" files. This is not intended to be the
1322 1       * number of files that resulted directly from the above
                                                          unmark_tree call.
1323 1       */

1325 1      if (rcp->rc_mark_summary_valid)
1326 2      {
1327 2           *fileMarked = rcp->rc_mark_summary.nfiles;
1328 2           *dirMarked = rcp->rc_mark_summary.ndirs;
1329 2           *otherMarked = rcp->rc_mark_summary.nother;
1330 1      }
1331 1      else
1332 2      {
1333 2           /*
1334 2            * This error is not fatal, the output variables *Marked,
1335 2            * will be zero, we should never get this error.
1336 2            */

1338 2           rbe_log_stats(
     0, "Internal error: mark summary not Valid in RSTSL_UnmarkObject() ");
1339 1      }

1341 1      if (save_time)
1342 1           RSTSL_SetBackupForTime( save_time, backup_flags );

1344 1      return( E_SUCCESS );
1345   }         /* end of RSTSL_UnmarkObject () */
```



```
1347   static void
1348   MarkUnmarkDebugLogEcho(tree_node *tnp)
1349 1 {
1350 1 #if defined(DEBUG_LOG_MARK_UNMK)
1351 2      {
1352 2           char pathbuf[EB_MAXPATHLEN];
1353 2           char *pbp = pathbuf;


1356 2           tn_getpath(tnp, &pbp);

1358 2           rbe_log_stats(0,"The file %s was marked", pbp);
1359 1      }
1360 1 #endif
1361 1 #if defined(DEBUG)
1362 2      {
1363 2           char pathbuf[EB_MAXPATHLEN];
1364 2           char *pbp = pathbuf;


1367 2           tn_getpath(tnp, &pbp);

1369 2           printf("The file %s was marked\n", pbp);
1370 1      }
1371 1 #endif
1372 1      return;
1373   }         /* MarkUnmarkDebugLogEcho */
```

D8

Fri Oct 10 14:49:35 2008 Function Index Page 2

Fri Oct 10 14:49:35 2008 Function Index Page 1

Fri Oct 10 14:49:35 2008 Function Index Page 2

```
/*****************************************************************************
                                                                          ******
**
** File Name:   RSTmedia.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**       This module contains the Restore API functions that provide
**       the information of the media needed for restore access. This
**       media list is updated in EDMRST_MarkObject()
**       and EDMRST_UnmarkObject().
**
** Table of Contents:
** ------------------
** public functions contained in:
**       EDMRST_GetNecessaryMedia
**       EDMRST_GetMediaVolid
**       EDMRST_GetMediaLabel
**       EDMRST_GetMediaSide
**      EDMRST_GetMediaSequenceNumber
**       EDMRST_GetMediaBarcodeString
**      EDMRST_GetMediaDescription
**       EDMRST_GetMediaStatus
**       EDMRST_GetMediaTrail
**      EDMRST_GetMediaLocation
**      EDMRST_GetMediaComments
**
**       EDMRST_GetNumberOfDuplicates
**      EDMRST_GetDupilcateVolid
**      EDMRST_GetDupilcateSequenceNumber
**       EDMRST_GetDupilcateBarcodeString
**      EDMRST_GetDupilcateTypeDescription
**      EDMRST_GetDupilcateTypeToken
**       EDMRST_GetDupilcateStatus
**       EDMRST_GetDupilcatetTrail
**      EDMRST_GetDupilcateLocation
**
** static functions NO LONGER contained here:
**       InitializeMediaObjects
**       AssignMediaObjects
**      MediaObjectConstructor
**       validateMediaObject
**       volid2str
**
**
** Compile-Time Options:
**              This section must list any compile time definitions
**              which will affect this header.
**
*****************************************************************************
                                                                      *****/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif
```



```
/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/ebutil.h>
#include <ebreport/ebvl.h>


/*
 * Local headers
 */
#include <RSTinterns.h>
#include <RSTsup_csm.h>


/*
 * #defines, structures, typedefs local to this source file
 */


/*
 * External declarations
 */


NEW_SRC_FILE();

/*
 * Local function prototypes
 */

static eerrno_ty CheckMediaObjects( const short maxEntries,
                                    mediaObject **objects );

static eerrno_ty copy_rpc_media_obj( mediaObject *dest,
                                     RSTRPC_media_object *src );
static eerrno_ty copy_rpc_media_dups( mediaObject *dest,
                                      RSTRPC_media_object *src );
```

```
/* public functions */

/*********************************************************************
 * Get Necessary Media:
 *
 *  This function is provided to allow retrieval of the
 *  necessary media to restore the currently marked objects.
 *
 *  The cookie must be initialize to INIT_COOKIE  on the first call to
 *  this routine.
                This routine will update the cookie to allow retrieval
 *  of more objects if there is more than "maxEntries".
                                                    The cookie will be
 *  returned as DONE_COOKIE when there are no more to retrieve.
 *
 * Parameters:
 *   svrHdl         (I)- a pointer to this user's client handle for the
 *                       Restore Engine (server) connection.
 *   maxEntries     (I)- the maximum number of media objects to return
 *   objects        (O)- an allocated array to return the objects in
 *   numberEntries (
          O)- the real number of media objects returned in the array
 *   cookie       (IO)- a place holder for the list position
 *
 *********************************************************************/

eerrno_ty
EDMRST_GetNecessaryMedia( serverHandle   svrHdl,
                          const short    maxEntries,
                           mediaObjectPtr *objects,
                           short          *numberEntries,
                           boolean_ty     all,
                           long           *cookie )
{
        RE_get_necessary_media_result   *rpc_result;
        RE_get_necessary_media_args     rpc_args;
        RSTRPC_media_list                *temp_list;
         eerrno_ty                       result = E_SUCCESS;

         /* validate inputs; */
         if (
          NULL == svrHdl || NULL == numberEntries || NULL == cookie ||
            maxEntries <= 0 )
                return EP_RB_RECOVER_BAD_ARGS;

         if (E_SUCCESS != (result = CheckMediaObjects( maxEntries,
                                                 (
                                            mediaObject **)objects)))
         {
             return result;
         }

        /* Prepare input argument structure for RPC: */
         rpc_args.maxEntries = maxEntries;
         rpc_args.cookie = *cookie;
         rpc_args.all = all;
         set_rpc_obj( re_get_necessary_media, &rpc_args.RPCobjID );

         rpc_result = re_get_necessary_media_1( &rpc_args, svrHdl );

         if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
         }
         else {
                if ( (result = rpc_result->status) == E_SUCCESS)
```



```
                {
                   *cookie = rpc_result->cookie;
                   *numberEntries = rpc_result->numEntries;
                   temp_list = rpc_result->mediaList;
                   while (
                      rpc_result->numEntries && result == E_SUCCESS )
                   {
                       if (
                    !temp_list || !*objects || !rpc_args.maxEntries--
                       || !temp_list->media_obj )
                              break;
                                 /* some null pointer or too many */
                      /* copy list object to array object entry: */
                       result = copy_rpc_media_obj( *objects,
                                                    temp_list->media_obj );
                       /* copy the duplicates for EACH media object
                        * into the media list stored in each
                                                                original
                        * media object
                        */
                       result = copy_rpc_media_dups( *objects++,
                                                    temp_list->media_obj );
                      temp_list = temp_list->next;
                      rpc_result->numEntries--;
                   }
                }

               /* release RPC result struct: */
                xdr_free( xdr_RE_get_necessary_media_result,
                          (char *)rpc_result );
        }

        return result;
}
```

```
211             /* EDMRST_GetNecessaryMedia */

213     static eerrno_ty
214     CheckMediaObjects(const short maxEntries,
215                       mediaObject **objects)
216   1 {
217   1     register int index;


220   1     if (NULL == objects)
221   2     {
222   2         return EP_RB_RECOVER_BAD_ARGS;
223   1     }

225   1     for (index = 0; index < maxEntries; index++)
226   2     {
227   2         if (NULL == objects[index]
228   2          || MEDIA_OBJECT != objects[index]->restoreObjType)
229   3         {
230   3             return EP_RB_RECOVER_BAD_ARGS;
231   2         }
232   1     }
233   1     return E_SUCCESS;
234     }       /* CheckMediaObjects */
```



```
237     static eerrno_ty copy_rpc_media_obj( mediaObject *dest,
238                                          RSTRPC_media_object *src )
239   1 {
240   1         if ( NULL == (dest->trail = esl_strdup( src->trail )) )
241   2         {
242   2                 rec_api_log_csm(SUB_CSM_NOMEM, NULL);
243   2                 return EP_RB_RECOVER_NOMEM;
244   1         }
245   1         if ( NULL == (dest->mtype = esl_strdup( src->mtype )) )
246   2         {
247   2                 rec_api_log_csm(SUB_CSM_NOMEM, NULL);
248   2                 return EP_RB_RECOVER_NOMEM;
249   1         }
250   1         if ( NULL == (dest->mtype_token = esl_strdup(
                                                    src->mtype_token )) )
251   2         {
252   2                 rec_api_log_csm(SUB_CSM_NOMEM, NULL);
253   2                 return EP_RB_RECOVER_NOMEM;
254   1         }
255   1         if ( NULL == (dest->barcode_label = esl_strdup(
                                                   src->barcode_label )) )
256   2         {
257   2                 rec_api_log_csm(SUB_CSM_NOMEM, NULL);
258   2                 return EP_RB_RECOVER_NOMEM;
259   1         }
260   1         if ( NULL == (dest->physical_loc = esl_strdup(
                                                    src->physical_loc )) )
261   2         {
262   2                 rec_api_log_csm(SUB_CSM_NOMEM, NULL);
263   2                 return EP_RB_RECOVER_NOMEM;
264   1         }
265   1         if ( NULL == (dest->comments = esl_strdup( src->comments )) )
266   2         {
267   2                 rec_api_log_csm(SUB_CSM_NOMEM, NULL);
268   2                 return EP_RB_RECOVER_NOMEM;
269   1         }
270   1         if ( NULL == (dest->volid_ascii = esl_strdup(
                                                    src->volid_ascii )) )
271   2         {
272   2                 rec_api_log_csm(SUB_CSM_NOMEM, NULL);
273   2                 return EP_RB_RECOVER_NOMEM;
274   1         }
275   1         if ( NULL == (dest->luname = esl_strdup( src->luname )) )
276   2         {
277   2                 rec_api_log_csm(SUB_CSM_NOMEM, NULL);
278   2                 return EP_RB_RECOVER_NOMEM;
279   1         }
280   1         dest->seqno = src->seqno;
281   1         dest->side = src->side;
282   1         dest->lmtime = src->lmtime;
283   1         dest->online = src->online;
284   1         dest->offsite = src->offsite;
285   1         dest->is_orig = src->is_orig;
286   1         dest->run_media_dup = src->run_media_dup;
287   1         /* added this to copy the number of duplicates */
288   1         dest->num_dups = src->num_dups;
289   1         return E_SUCCESS;
290     }
```

```
294  /*******************************************************
295   * copy_rpc_media_dups
296   * goes through the list of duplicates from the RSTRPC_media_object
297   * and calls the copy media obj function to copy the fields into
298   * the new structure.  This just copies the linked list of duplicates
299   *******************************************************
300   */
301  static eerrno_ty
302  copy_rpc_media_dups( mediaObject *dest,
303                       struct RSTRPC_media_object *src )
304 1{
305
306 1struct mediaObjectList   *dst_list_pointer;
                                /* The list of objects to be copied to */
307 1 struct RSTRPC_media_list *src_list_pointer;
                                /* List of objects to be copied from */
308 1eerrno_ty                result=E_SUCCESS;
309
310 1 dst->dups=calloc(1,sizeof(struct mediaObjectList));
                                /* creates the first list item  */
311
312 1dst_list_pointer = dst->dups;
313
314 1src_list_pointer = src->dups;
315
316 1/* traverse the source media list */
317 1while(src_list_pointer!=NULL)
318 2  {
319 2    dst_list_pointer->media_obj = calloc(1,sizeof(mediaObject));
320 2    result = copy_rpc_media_obj(
              dst_list_pointer->media_obj, src_list_pointer->media_obj);
321 2    if (result != E_SUCCESS)  /* if the copy had an error lets exit */
322 2      return result;
323
324 2    src_list_pointer=src_list_pointer->next;
                                /* move to the next media object */
325
326 2    if (src_list_pointer != NULL)  /* if we still have more to copy */
327 3      {
328 3         dst_list_pointer->next=calloc(1,sizeof(
                                        struct mediaObjectList));
329 3         dst_list_pointer=dst_list_pointer->next;
330 2      }
331 2    else  /* no more to copy */
332 3      {
333 3         dst_list_pointer->next=NULL;
334 2      }
335 1  }
336 1return result;
337  }
```



```
339  /*******************************************************************
340   * Media Object Access Routines:
341   *
342   *  These routines retrieve information pertinent to a given Media
                                                                  object.
343   *
344   * Parameters:
345   *
346   *  svrHdl      (I) - (
                    ignored) A pointer to this user's client handle for the
347   *                    Restore Engine (server) connection.
348   *  thisObject (I) - The media object
349   * For the duplicate functions
350   *  dup_number (
                 I) - The number of the duplicate to retrieve from usually
351   *                    1 for now, until multiple duplicates can be made
352   *
353   * RETURNS one of the following:
354   *       const char *    pointer to a string within the media object,
                                                                  that should
355   *                       not be changed.
356   *       MediaStatus
357   *       long            media sequence number
358   *       uchar_t                 media side
359   *
360   *******************************************************************/
361
362  const char *
363  EDMRST_GetMediaVolid( serverHandle   svrHdl,
364                        mediaObjectPtr thisObject )
365 1{
366 1    if ( (NULL == svrHdl) || (NULL == thisObject)
367 1     || (NULL == handlePtr) || (
                                  svrHdl != handlePtr->re_binding_handle)
368 1     || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
369 1    )
370 1         return NULL;
371
372 1    return ((mediaObject *)thisObject)->volid_ascii;
373  }       /* EDMRST_GetMediaVolid */
```

```
375   const char *
376   EDMRST_GetMediaLUName( serverHandle   svrHdl,
377                          mediaObjectPtr thisObject )
378 1 {
379 1     if ( (NULL == svrHdl) || (NULL == thisObject)
380 1      || (NULL == handlePtr) || (
                                   svrHdl != handlePtr->re_binding_handle)
381 1      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
382 1     )
383 1          return NULL;

385 1     return ((mediaObject *)thisObject)->luname;
386   }         /* EDMRST_GetMediaVolid */
```



```
388   const char *
389   EDMRST_GetMediaTrail( serverHandle   svrHdl,
390                         mediaObjectPtr thisObject )
391 1 {
392 1     if ( (NULL == svrHdl) || (NULL == thisObject)
393 1      || (NULL == handlePtr) || (
                                   svrHdl != handlePtr->re_binding_handle)
394 1      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
395 1     )
396 1          return NULL;

398 1     return ((mediaObject *)thisObject)->trail;
399   }         /* EDMRST_GetMediaTrail */
```

```
401   uchar_t
402   EDMRST_GetMediaSide( serverHandle   svrHdl,
403                        mediaObjectPtr thisObject )
404 1 {
405 1     if ( (NULL == svrHdl) || (NULL == thisObject)
406 1       || (NULL == handlePtr) || (
                                  svrHdl != handlePtr->re_binding_handle)
407 1       || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
408 1     )
409 1           return 0;

411 1     return ((mediaObject *)thisObject)->side;
412   }         /* EDMRST_GetMediaSide */
```



```
414   long
415   EDMRST_GetMediaSequenceNumber( serverHandle   svrHdl,
416                                  mediaObjectPtr thisObject )
417 1 {
418 1     if ( (NULL == svrHdl) || (NULL == thisObject)
419 1       || (NULL == handlePtr) || (
                                  svrHdl != handlePtr->re_binding_handle)
420 1       || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
421 1     )
422 1           return 0;

424 1     return (long)((mediaObject *)thisObject)->seqno;
425   }         /* EDMRST_GetMediaSequenceNumber */
```

```
427   const char *
428   EDMRST_GetMediaBarcodeString( serverHandle   svrHdl,
429                                 mediaObjectPtr thisObject )
430 1 {
431 1     if ( (NULL == svrHdl) || (NULL == thisObject)
432 1      || (NULL == handlePtr) || (
                                    svrHdl != handlePtr->re_binding_handle)
433 1      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
434 1     )
435 1          return NULL;

437 1     return ((mediaObject *)thisObject)->barcode_label;
438   }         /* EDMRST_GetMediaBarcodeString */
```



```
440   const char *
441   EDMRST_GetMediaTypeDescrip( serverHandle   svrHdl,
442                               mediaObjectPtr thisObject )
443 1 {
444 1     if ( (NULL == svrHdl) || (NULL == thisObject)
445 1      || (NULL == handlePtr) || (
                                    svrHdl != handlePtr->re_binding_handle)
446 1      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
447 1     )
448 1          return NULL;

450 1     return ((mediaObject *)thisObject)->mtype;
451   }         /* EDMRST_GetMediaTypeDescrip */
```

```
453     const char *
454     EDMRST_GetMediaTypeToken( serverHandle   svrHdl,
455                               mediaObjectPtr thisObject )
456   1 {
457   1     if ( (NULL == svrHdl) || (NULL == thisObject)
458   1      || (NULL == handlePtr) || (
459   1                                svrHdl != handlePtr->re_binding_handle)
460   1      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
461   1     )
463   1          return NULL;

464   1     return ((mediaObject *)thisObject)->mtype_token;
465   }         /* EDMRST_GetMediaTypeToken */
```



```
466     MediaStatus
467     EDMRST_GetMediaStatus( serverHandle   svrHdl,
468                            mediaObjectPtr thisObject )
469   1 {
470   1     if ( (NULL == svrHdl) || (NULL == thisObject)
471   1      || (NULL == handlePtr) || (
472   1                                svrHdl != handlePtr->re_binding_handle)
473   1      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
474   1     )
476   1          return Media_Offline;

477   1     if (((mediaObject *)thisObject)->online)
478   2     {
479   2          return Media_Online;
480   1     }
481   1     else if (!(((mediaObject *)thisObject)->offsite))
482   2     {
483   2          /*
484   2           * offline & onsite
485   2           */

486   2          return Media_Offline;
487   1     }
488   1     else
489   2     {
490   2          /*
491   2           * offsite & offline
492   2           */

494   2          return Media_Offsite;
495   1     }
496   }         /* EDMRST_GetMediaStatus */
```

```
498     const char *
499     EDMRST_GetMediaLocation( serverHandle   svrHdl,
500                              mediaObjectPtr thisObject )
501   1 {
502   1     if ( (NULL == svrHdl) || (NULL == thisObject)
503   1      || (NULL == handlePtr) || (
                                    svrHdl != handlePtr->re_binding_handle)
504   1      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
505   1     )
506   1          return NULL;

508   1     return ((mediaObject *)thisObject)->physical_loc;
509     }         /* EDMRST_GetMediaLocation */
```



```
511     const char *
512     EDMRST_GetMediaComments( serverHandle   svrHdl,
513                              mediaObjectPtr thisObject )
514   1 {
515   1     if ( (NULL == svrHdl) || (NULL == thisObject)
516   1      || (NULL == handlePtr) || (
                                    svrHdl != handlePtr->re_binding_handle)
517   1      || (MEDIA_OBJECT != ((mediaObject *)thisObject)->restoreObjType)
518   1     )
519   1          return NULL;

521   1     return ((mediaObject *)thisObject)->comments;
522     }         /* EDMRST_GetMediaComments */
```

```
525  /*****************************************************
526   * Duplicate Media Access Routines
527   * Inputs: Svr Handle - see above
528   *         dup_number: the number of the duplicate wanted
529   *         thiObject:  The media object to get the dups for...
530   *****************************************************
531   */
532  short EDMRST_GetNumberOfDuplicates( serverHandle   svrHdl,
533                                      mediaObjectPtr thisObject )
534  1 {
535  1   mediaObject *tempObj;
536  1   tempObj = (mediaObject *)thisObject;
537  1   return tempObj->num_dups;
538    }
```



```
542  const char *
543  EDMRST_GetDuplicateVolid( serverHandle   svrHdl,
544                            int dup_number,
545                            mediaObjectPtr thisObject )
546  1 {
547  1   mediaObject       *dupObject;
548  1   struct mediaObjectList   *dupObjectList;
549  1   short curr_dup=0;

551  1   dupObject = (mediaObject *)thisObject; /* kinda cheating here,
552  1                                           * this is the original, but
553  1                                           * the variable becomes the
554  1                                           * duplicate  further down
                                                                          */
555  1   dupObjectList = dupObject->dups;  /* Points to First mediaObject */

557  1   /* Make sure we have something to work with */
558  1   if ( (NULL == svrHdl) || (NULL == thisObject)
559  1        || (dup_number >dupObject->num_dups)
560  1        || (NULL == handlePtr) || (
                                        svrHdl != handlePtr->re_binding_handle)
561  1        || (MEDIA_OBJECT != (dupObject->restoreObjType)))
562  1     return NULL;

564  1   /* get to specified Object, but already at first one */
565  1   for (curr_dup=1; curr_dup<dup_number; curr_dup++)
566  2     {
567  2       dupObjectList = dupObjectList->next;
568  1     }

570  1   /* get the media object */
571  1   dupObject = (mediaObject *) dupObjectList->media_obj;

573  1   /* return the volid */
574  1   return dupObject->volid_ascii;
575  1          /* EDMRST_GetDuplicateVolid */
576    }
```

```
580   const char *
581   EDMRST_GetDuplicateLUName( serverHandle   svrHdl,
582                              int dup_number,
583                              mediaObjectPtr thisObject )
584 1 {
585 1   mediaObject       *dupObject;
586 1   struct mediaObjectList   *dupObjectList;
587 1   short curr_dup=0;

589 1   dupObject = (mediaObject *)thisObject; /* kinda cheating here,
590 1                                          * this is the original, but
591 1                                          * the variable becomes the
592 1                                          * duplicate  further down
                                                                      */
593 1   dupObjectList = dupObject->dups;  /* Points to First mediaObject */

595 1   /* Make sure we have something to work with */
596 1   if ( (NULL == svrHdl) || (NULL == thisObject)
597 1        || (dup_number >dupObject->num_dups)
598 1        || (NULL == handlePtr) || (
                                    svrHdl != handlePtr->re_binding_handle)
599 1        || (MEDIA_OBJECT != (dupObject->restoreObjType)))
600 1     return NULL;

602 1   /* get to specified Object, but already at first one */
603 1   for (curr_dup=1; curr_dup<dup_number; curr_dup++)
604 2     {
605 2       dupObjectList = dupObjectList->next;
606 1     }

608 1   /* get the media object */
609 1   dupObject = (mediaObject *) dupObjectList->media_obj;

611 1   /* return the volid */
612 1   return dupObject->luname;
613 1          /* EDMRST_GetDuplicateVolid */
614   }
```



```
616   long
617   EDMRST_GetDuplicateSequenceNumber( serverHandle   svrHdl,
618                                      int dup_number,
619                                      mediaObjectPtr thisObject )
620 1 {
621 1   mediaObject       *dupObject;
622 1   struct mediaObjectList   *dupObjectList;
623 1   short curr_dup=0;

625 1   dupObject = (mediaObject *)thisObject; /* kinda cheating here,
626 1                                          * this is the original, but
627 1                                          * the variable becomes the
628 1                                          * duplicate  further down
                                                                      */
629 1   dupObjectList = dupObject->dups; /* Points to First mediaObject */

631 1   /* Make sure we have something to work with */
632 1   if ( (NULL == svrHdl) || (NULL == thisObject)
633 1        || (dup_number >dupObject->num_dups)
634 1        || (NULL == handlePtr) || (
                                    svrHdl != handlePtr->re_binding_handle)
635 1        || (MEDIA_OBJECT != (dupObject->restoreObjType)))
636 1     return 0;

638 1   /* get to specified Object, but already at first one */
639 1   for (curr_dup=1;curr_dup<dup_number; curr_dup++)
640 2     {
641 2       dupObjectList = dupObjectList->next;
642 1     }

644 1   dupObject = (mediaObject *) dupObjectList->media_obj;

646 1   return dupObject->seqno;
647   }         /* EDMRST_GetDuplicateSequenceNumber */
```

```
652   const char *
653   EDMRST_GetDuplicateBarcodeString( serverHandle   svrHdl,
654                                     int dup_number,
655                                     mediaObjectPtr thisObject )
656 1 {
657 1   mediaObject       *dupObject;
658 1   struct mediaObjectList   *dupObjectList;
659 1   short curr_dup=0;

661 1   dupObject = (mediaObject *)thisObject; /* kinda cheating here,
662 1                                          * this is the original, but
663 1                                          * the variable becomes the
664 1                                          * duplicate  further down
                                                                      */
665 1   dupObjectList = dupObject->dups; /* Points to First mediaObject */

667 1   /* Make sure we have something to work with */
668 1   if ( (NULL == svrHdl) || (NULL == thisObject)
669 1        || (dup_number >dupObject->num_dups)
670 1        || (NULL == handlePtr) || (
                                    svrHdl != handlePtr->re_binding_handle)
671 1        || (MEDIA_OBJECT != (dupObject->restoreObjType)))
672 1     return NULL;

674 1   /* get to specified Object, but already at first one */
675 1   for (curr_dup=1;curr_dup<dup_number; curr_dup++)
676 2     {
677 2       dupObjectList = dupObjectList->next;
678 1     }

680 1   dupObject = (mediaObject *) dupObjectList->media_obj;

682 1   return dupObject->barcode_label;
683   }       /* EDMRST_GetDuplicateBarcodeString */
```



```
689   const char *
690   EDMRST_GetDuplicateTypeDescrip( serverHandle   svrHdl,
691                                   int dup_number,
692                                   mediaObjectPtr thisObject )
693 1 {
694 1     mediaObject       *dupObject;
695 1   struct mediaObjectList   *dupObjectList;
696 1   short curr_dup=0;

698 1   dupObject = (mediaObject *)thisObject; /* kinda cheating here,
699 1                                          * this is the original, but
700 1                                          * the variable becomes the
701 1                                          * duplicate  further down
                                                                      */
702 1   dupObjectList = dupObject->dups; /* Points to First mediaObject */

704 1   /* Make sure we have something to work with */
705 1   if ( (NULL == svrHdl) || (NULL == thisObject)
706 1        || (dup_number >dupObject->num_dups)
707 1        || (NULL == handlePtr) || (
                                    svrHdl != handlePtr->re_binding_handle)
708 1        || (MEDIA_OBJECT != (dupObject->restoreObjType)))
709 1     return NULL;

711 1   /* get to specified Object, but already at first one */
712 1   for (curr_dup=1;curr_dup<dup_number; curr_dup++)
713 2     {
714 2       dupObjectList = dupObjectList->next;
715 1     }

717 1   dupObject = (mediaObject *) dupObjectList->media_obj;

719 1   return dupObject->mtype;
720   }       /* EDMRST_GetDuplicateTypeDescrip */
```

```
726     const char *
727     EDMRST_GetDuplicateTypeToken( serverHandle   svrHdl,
728                                   int dup_number,
729                                   mediaObjectPtr thisObject )
730   1 {
731   1   mediaObject       *dupObject;
732   1   struct mediaObjectList   *dupObjectList;
733   1   short curr_dup=0;
734   1
735   1   dupObject = (mediaObject *)thisObject; /* kinda cheating here,
736   1                                           * this is the original, but
737   1                                           * the variable becomes the
738   1                                           * duplicate  further down
739   1                                           */
740   1   dupObjectList = dupObject->dups; /* Points to First mediaObject */
741   1
742   1   /* Make sure we have something to work with */
743   1   if ( (NULL == svrHdl) || (NULL == thisObject)
744   1        || (dup_number >dupObject->num_dups)
745   1        || (NULL == handlePtr) || (
746   1                                   svrHdl != handlePtr->re_binding_handle)
747   1        || (MEDIA_OBJECT != (dupObject->restoreObjType)))
748   1     return NULL;
749   1
750   1   /* get to specified Object, but already at first one */
751   1   for (curr_dup=1;curr_dup<dup_number; curr_dup++)
752   2     {
753   2       dupObjectList = dupObjectList->next;
754   1     }
755   1
756   1   dupObject = (mediaObject *) dupObjectList->media_obj;
757   1
758   1   return dupObject->mtype_token;
759   1
760     }         /* EDMRST_GeDuplicatetTypeToken */
```



```
763     MediaStatus
764     EDMRST_GetDuplicateStatus( serverHandle   svrHdl,
765                                int dup_number,
766                                mediaObjectPtr thisObject )
767   1 {
768   1   mediaObject       *dupObject;
769   1   struct mediaObjectList   *dupObjectList;
770   1   short curr_dup=0;
771   1
772   1   dupObject = (mediaObject *)thisObject; /* kinda cheating here,
773   1                                           * this is the original, but
774   1                                           * the variable becomes the
775   1                                           * duplicate  further down
776   1                                           */
777   1   dupObjectList = dupObject->dups; /* Points to First mediaObject */
778   1
779   1   /* Make sure we have something to work with */
780   1   if ( (NULL == svrHdl) || (NULL == thisObject)
781   1        || (dup_number >dupObject->num_dups)
782   1        || (NULL == handlePtr) || (
783   1                                   svrHdl != handlePtr->re_binding_handle)
784   1        || (MEDIA_OBJECT != (dupObject->restoreObjType)))
785   1       return Media_Offline;
786   1
787   1   /* get to specified Object, but already at first one */
788   1   for (curr_dup=1;curr_dup<dup_number; curr_dup++)
789   2     {
790   2       dupObjectList = dupObjectList->next;
791   1     }
792   1
793   1   dupObject = (mediaObject *) dupObjectList->media_obj;
794   1
795   1     if (dupObject->online)
796   2     {
797   2          return Media_Online;
798   1     }
799   1     else if (!(dupObject->offsite))
800   2     {
801   2          /*
802   2           * offline & onsite
803   2           */
804   2
805   2          return Media_Offline;
806   1     }
807   1     else
808   2     {
809   2          /*
810   2           * offsite & offline
811   2           */
812   2
813   2          return Media_Offsite;
814   1     }
815     }         /* EDMRST_GetDuplicateStatus */
```

```
819   const char *
820   EDMRST_GetDuplicateLocation( serverHandle   svrHdl,
821                                int dup_number,
822                                mediaObjectPtr thisObject )
823 1 {
824 1   mediaObject       *dupObject;
825 1   struct mediaObjectList   *dupObjectList;
826 1   short curr_dup=0;
827 1
828 1   dupObject = (mediaObject *)thisObject; /* kinda cheating here,
829 1                                          * this is the original, but
830 1                                          * the variable becomes the
831 1                                          * duplicate  further down
832 1                                                                */
833 1   dupObjectList = dupObject->dups; /* Points to First mediaObject */
834 1
835 1   /* Make sure we have something to work with */
836 1   if ( (NULL == svrHdl) || (NULL == thisObject)
837 1        || (dup_number >dupObject->num_dups)
838 1        || (NULL == handlePtr) || (
                                        svrHdl != handlePtr->re_binding_handle)
839 1        || (MEDIA_OBJECT != (dupObject->restoreObjType)))
840 1     return NULL;
841 1
842 1   /* get to specified Object, but already at first one */
843 1   for (curr_dup=1;curr_dup<dup_number; curr_dup++)
844 2     {
845 2       dupObjectList = dupObjectList->next;
846 1     }
847 1
848 1   dupObject = (mediaObject *) dupObjectList->media_obj;
849 1
850 1   return dupObject->physical_loc;
851   }         /* EDMRST_GetDuplicateLocation */
```

Fri Oct 10 14:50:18 2008 Function Index Page 2

Fri Oct 10 14:50:18 2008 Function Index Page 1

Fri Oct 10 14:50:18 2008 Function Index Page 2

```
/*****************************************************************
                                                          ******
**
** File Name:   RSLmedia.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**       This module contains the Restore Service Library function
                                                        that provides
**       the information of the media needed for restore access. This
**       media list is updated in EDMRST_MarkObject(
                                        ) and EDMRST_UnmarkObject().
**
** Table of Contents:
** ------------------
** public functions contained in:
**       RSTSL_GetNecessaryMedia
**
** static functions contained here:
**       AssignMediaObjects
**      MediaObjectConstructor
**       validateMediaObject
**       volid2str
**
**
** Compile-Time Options:
**              This section must list any compile time definitions
**              which will affect this header.
**
*****************************************************************
                                                          *****/



/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif



/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1
#define  MAX_DUPS 256

/*
 * System headers.
 */


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
```



```
#include <eb/rb_log.h>
#include <ebutil/ebutil.h>
#include <ebreport/ebvl.h>


/*
 * Local headers
 */
#include <RSLinterns.h>


/*
 * #defines, structures, typedefs local to this source file
 */


/*
 * External declarations
 */


NEW_SRC_FILE();

/*
 * Local function prototypes
 */

static eerrno_ty AssignMediaObjects(
                                const ebvl_volidlist_ty      *Init_volume,
                                    const short              maxEntries,
                                    struct RSTRPC_media_list
                                                               **objects,
                                    short
                                                            *numberEntries,
                                    boolean_ty                 all,
                                    long                       *cookie );

static eerrno_ty MediaObjectConstructor(
                                     const ebvl_volidlist_ty
                                                          *Internal_volume,
                                     struct RSTRPC_media_object
                                                         **External_volume,
                                     boolean_ty GetOriginalVolume);

static void validateMediaObject( volumeID_ty *volid,
                                      struct RSTRPC_media_object
                                                           *thisObject );

static char *volid2str(const volumeID_ty *volid);


/*
 * Used for debugging
 */

eerrno_ty loadMediaObject(mediaObject *MedObjPtr);
                                                    /* Used for debugging */
void GetMediaDebugLogEcho(const mediaObject *MedObjPtr);
```

```
114  /* public functions */
116  /**********************************************************************
117   * Get Necessary Media:
118   *
119   *  This function is provided to allow retrieval of the
120   *  necessary media to restore the currently marked objects.
121   *
122   *  The cookie must be initialize to INIT_COOKIE  on the first call to
123   *  this routine.
                    This routine will update the cookie to allow retrieval
124   *  of more objects if there is more than "maxEntries".
                                                    The cookie will be
125   *  returned as DONE_COOKIE when there are no more to retrieve.
126   *
127   * Parameters:
128   *   maxEntries     (I)- the maximum number of media objects to return
129   *   objects        (
              O)- pointer to a linked list of the returned media objects
130   *   numberEntries (
               O)- the real number of media objects returned in the list
131   *   cookie        (IO)- a place holder for the list position
132   *
133   **********************************************************************/
135  eerrno_ty
136  RSTSL_GetNecessaryMedia( const short    maxEntries,
137                           struct RSTRPC_media_list **objects,
138                           short          *numberEntries,
139                           boolean_ty     all,
140                           long           *cookie )
141 1 {
142 1      static long     valid_cookie;
                                /* Used to validate incoming cookies */
143 1      static long     valid_dups_cookie;
                                /* Used to validate incoming cookies */
144 1      short           index = 0;
145 1      eerrno_ty       ep_status = E_SUCCESS;
147 1      short count = 0;
148 1      struct RSTRPC_media_list *tmp_list = NULL;
149 1      struct RSTRPC_media_list *end_list = NULL;
151 1      if ( NULL==objects || NULL==numberEntries || NULL==cookie ||
152 1           maxEntries <= 0 )
153 1           return EP_RB_RECOVER_BAD_ARGS;
155 1      if (INIT_COOKIE == *cookie)
156 2      {
157 2          /*
158 2            * This is the initial call to EDMRST_GetNecessaryMedia ()
159 2            */
161 2          /* Check if this is a plugin's TLO and if it has its own
                                                                  func: */
162 2          if ( 0 != rcp-> rc_backup_app
163 2            && (((struct pluginIDdata *)(
                                rcp-> currentPIptr-> idData))-> options
164 2             & RSTPI_OPTION_MASK_GET_MEDIA ) ==
                                        RSTPI_OPTION_SPECIAL_GET_MEDIA )
165 3          {
166 3              ep_status = rcp-> currentPIptr-> piFuncArray[
                                                  PIFuncIndexGetMedia ]
167 3                                                (
                                               rcp, &tmp_list, &count);
168 3              if (E_SUCCESS == ep_status)
```



```
169 4              {
170 4                  *objects = tmp_list;    /* give them start of list */
171 4                  if (count <= maxEntries)
172 5                  {
173 5                          *numberEntries = count;
174 5                          valid_cookie = DONE_COOKIE;
175 4                  }
176 4                  else
177 5                  { /* return max or to end of list */
178 5                          for ( *numberEntries = 0;
179 5                                *numberEntries < maxEntries && tmp_list;
180 5                                (
                                *numberEntries)++, tmp_list = tmp_list->next
181 5                                )
182 5                                  end_list = tmp_list;
184 6                          if (
                        tmp_list) {                 /* terminate returned list */
185 6                                  end_list->next = NULL;
186 6                                  valid_cookie = (long)tmp_list;
187 5                          }
188 5                          else
189 5                                  valid_cookie = DONE_COOKIE;
190 4                  }
191 3              }
192 2          }
194 2          else    /* othwrwise, plugin uses generic logic: */
195 2              ep_status = AssignMediaObjects( rcp -> ebvllist,
196 2                                              maxEntries,
197 2                                              objects,
198 2                                              numberEntries,
199 2                                              all,
200 2                                              &valid_cookie );
202 1      }
203 1      else if ((DONE_COOKIE == *cookie) || (valid_cookie != *cookie))
204 2      {
205 2          /*
206 2           * Invalid cookie
207 2           */
209 2          return EP_RB_RECOVER_BAD_COOKIE;
210 1      }
211 1      else
212 2      {
213 2          /*
214 2           * This is a follow up call to EDMRST_GetNecessaryMedia ()
215 2           */
217 2          /* Check if this is a plugin's TLO and if it has its own
                                                                  func: */
219 2          if ( 0 != rcp-> rc_backup_app
220 2            && ((struct pluginIDdata *)(
                                rcp-> currentPIptr-> idData))-> options
221 2               & RSTPI_OPTION_MASK_GET_MEDIA ==
                                        RSTPI_OPTION_SPECIAL_GET_MEDIA )
222 3          {
223 3          /* return max or to end of list */
224 3                  tmp_list = (struct RSTRPC_media_list *) valid_cookie;
225 3                  for ( *numberEntries = 0, *objects = tmp_list;
226 3                        *numberEntries < maxEntries && tmp_list;
227 3                        (*numberEntries)++, tmp_list = tmp_list->next
228 3                  )
229 3                          end_list = tmp_list;
```

```
231 4                   if (
                    tmp_list) {                 /* terminate returned list */
232 4                           end_list->next = NULL;
233 4                           valid_cookie = (long)tmp_list;
234 3                   }
235 3                   else
236 3                           valid_cookie = DONE_COOKIE;
237 2           }

239 2           else
240 2                   ep_status = AssignMediaObjects( (
                                        ebvl_volidlist_ty *) *cookie,
241 2                                           maxEntries,
242 2                                           objects,
243 2                                           numberEntries,
244 2                                           all,
245 2                                           &valid_cookie );
246 1   }

248 1   if (ep_status == E_SUCCESS)
249 1       *cookie = valid_cookie;

251 1   return( ep_status );
252 }       /* RSTSL_GetNecessaryMedia */
```



```
255 /*************************************************************
256  * MediaObjectConstructor ()
257  *
258  * Assign fields to MediaObject from ebvl_volidlist_ty.
259  * The (char *) are malloc'ed.
260  *
261  * Returns: E_SUCCESS for success, Epoch Extended Errnum for Failure;
262  *       EP_RB_RECOVER_INVAL_OBJTYPE     if volid is 0
263  *       EP_RB_RECOVER_BAD_ARGS          if Internal_volume is 0
264  *       EP_RB_RECOVER_NOMEM             if malloc fails
265  *
266  * Parameters:
267  *
268  *   Internal_volume   (I)- internal volumes ebvl_volidlist_ty.
269  *   External_volume   (O)- pointer to an allocated Media object with (
                                                                  char *)
270  *                        fields allocated and assigned.
271  *
272  *************************************************************/

274 static eerrno_ty
275 MediaObjectConstructor(
                        const ebvl_volidlist_ty          *Internal_volume,
276                         struct RSTRPC_media_object
                                                        **External_volume,
277                         boolean_ty
                                                         GetOriginalVolume)
278 1 {
279 1     const ebvl_volidlist_ty *IntVol = Internal_volume;
280 1     struct RSTRPC_media_object *ExtVol
281 1                         = calloc( 1, sizeof (
                                            struct RSTRPC_media_object) );
282 1     eerrno_ty    result = E_SUCCESS;

284 1     /*
285 1      * Complain if the given list of volumes is null.
286 1      */

288 1     if (NULL == IntVol)
289 2     {
290 2         rbe_log_stats(
                        EINVAL, "NULL Internal_volume in %s at line %d",
291 2                      __FILE__, __LINE__);
292 2         return EP_RB_RECOVER_BAD_ARGS;
293 1     }

295 1     if (NULL == ExtVol)
296 2     {
297 2         rbe_log_stats(
                    ENOMEM, "Cannot allocate media object in %s at line %d",
298 2                      __FILE__, __LINE__);
299 2         return EP_RB_RECOVER_NOMEM;
300 1     }

302 1     /*
303 1      * Find the first volume in this volume group which is "listable".
304 1      *
305 1      * For now (
                    11/13/97), ebrecover_api will pay attention only to the
306 1      * first listable volume in the volume group.
                                                        If there are multiple
307 1      * listable volumes in the volume group, the others are ignored.
308 1      * This issue is intended to be addressed in the future.
309 1      */
```

```
311 1      if ( FALSE == GetOriginalVolume )
312 2       {
313 2            for (; NULL != IntVol && !IntVol->preferred;
                                                    IntVol = IntVol->dupsP)
314 3             {
315 3                  /* null */
316 2             }
317 1       }

319 1      /*
320 1       * If none were "listable" [should not happen],
321 1       * just work with the original.
322 1       */

324 1      if (NULL == IntVol)
325 2      {
326 2           IntVol = Internal_volume;
327 1      }

329 2      do {          /* just to allow break on malloc failures */
330 2 #if 0
331 2      /*
332 2       * Standard copy volumeID_ty of type
333 2       */

335 2      ExtVol -> volid.opaque[0] = IntVol -> volid.opaque[0];
336 2      ExtVol -> volid.opaque[1] = IntVol -> volid.opaque[1];
337 2 #endif
338 2           ExtVol -> volid_ascii = esl_strdup(volid2str(&(
                                                    IntVol -> volid)));
339 2           if (NULL == ExtVol->volid_ascii)
340 3           {
341 3                rec_api_log_csm(SUB_CSM_NOMEM, NULL);
342 3                result = EP_RB_RECOVER_NOMEM;
343 3                break;
344 2           }

346 2           if (!(IntVol -> volid.opaque[0]) && !(
                                          IntVol -> volid.opaque[1]))
347 3           {
348 3                /*
349 3                 * The volid == 0000000000000000 should be logged
350 3                 */

352 3                rbe_log_stats( 0,"Internal Error: VOLID is invalid %s",
353 3                          ExtVol -> volid_ascii);
354 3                result = EP_RB_RECOVER_INVAL_OBJTYPE;
355 3                break;
356 2           }

358 2           if (IntVol -> trail)
359 3           {
360 3                ExtVol -> trail = esl_strdup(IntVol -> trail);
361 3                if (NULL == ExtVol->trail)
362 4               {
363 4                     rec_api_log_csm(SUB_CSM_NOMEM, NULL);
364 4                     result = EP_RB_RECOVER_NOMEM;
365 4                     break;
366 3               }
367 2           }
368 2           if (IntVol -> luname)
369 3           {
370 3                ExtVol -> luname = esl_strdup(IntVol -> luname);
371 3                if (NULL == ExtVol->luname)
372 4               {
```



```
373 4                     rec_api_log_csm(SUB_CSM_NOMEM, NULL);
374 4                     result = EP_RB_RECOVER_NOMEM;
375 4                     break;
376 3                }
377 2           }

379 2           if (IntVol -> mtype)
380 3           {
381 3                ExtVol -> mtype =  esl_strdup(IntVol -> mtype);
382 3                if (NULL == ExtVol->mtype)
383 4               {
384 4                     rec_api_log_csm(SUB_CSM_NOMEM, NULL);
385 4                     result = EP_RB_RECOVER_NOMEM;
386 4                     break;
387 3                }
388 2           }

390 2           if (IntVol -> mtype_token)
391 3           {
392 3                ExtVol -> mtype_token = esl_strdup(IntVol -> mtype_token);
393 3                if (NULL == ExtVol->mtype_token)
394 4               {
395 4                     rec_api_log_csm(SUB_CSM_NOMEM, NULL);
396 4                     result = EP_RB_RECOVER_NOMEM;
397 4                     break;
398 3                }
399 2           }

401 2           if (IntVol -> barcode_label)
402 3           {
403 3                ExtVol -> barcode_label = esl_strdup(
                                                    IntVol -> barcode_label);
404 3                if (NULL == ExtVol->barcode_label)
405 4               {
406 4                     rec_api_log_csm(SUB_CSM_NOMEM, NULL);
407 4                     result = EP_RB_RECOVER_NOMEM;
408 4                     break;
409 3                }
410 2           }

412 2           if (IntVol -> physloc)
413 3           {
414 3                ExtVol -> physical_loc = esl_strdup(IntVol -> physloc);
415 3                if (NULL == ExtVol->physical_loc)
416 4               {
417 4                     rec_api_log_csm(SUB_CSM_NOMEM, NULL);
418 4                     result = EP_RB_RECOVER_NOMEM;
419 4                     break;
420 3                }
421 2           }

423 2            if (IntVol -> comment)
424 3            {
425 3                    ExtVol -> comments = esl_strdup(IntVol -> comment);
426 3                    if (NULL == ExtVol->comments)
427 4                    {
428 4                        rec_api_log_csm(SUB_CSM_NOMEM, NULL);
429 4                        result = EP_RB_RECOVER_NOMEM;
430 4                        break;
431 3                    }
432 2            }

434 1      } while (0);                  /* end do */

436 2      if (result != E_SUCCESS) {
437 2           FreeMediaObjectContents( ExtVol );
```

```
438 2          free( ExtVol );
439 2           return result;
440 1      }

442 1      ExtVol -> seqno = IntVol -> seqno;

444 1      switch (IntVol -> side)
445 2      {
446 2           case'a':
447 2           case'A':
448 2               ExtVol -> side = 'A';
449 2               break;

451 2           case'b':
452 2           case'B':
453 2                ExtVol -> side = 'B';
454 2                break;

456 2          default:
457 2                ExtVol -> side = '\0';
458 1      }

460 1      if (FALSE == IntVol -> online)
461 2      {
462 2           ExtVol -> online = FALSE;
463 1      }
464 1      else
465 2      {
466 2           ExtVol -> online = TRUE;
467 1      }

469 1      if (FALSE == IntVol -> offsite)
470 2      {
471 2           ExtVol -> offsite = FALSE;
472 1      }
473 1      else
474 2      {
475 2           ExtVol -> offsite = TRUE;
476 1      }

478 1      if (FALSE == IntVol -> is_orig)
479 2      {
480 2           ExtVol -> is_orig = FALSE;
481 1      }
482 1      else
483 2      {
484 2           ExtVol -> is_orig = TRUE;
485 1      }

487 1      if (FALSE == IntVol -> run_media_dup)
488 2      {
489 2           ExtVol -> run_media_dup = FALSE;
490 1      }
491 1      else
492 2      {
493 2           ExtVol -> run_media_dup = TRUE;
494 1      }

496 1      ExtVol -> dups = NULL;
497 1 #if 0
498 1      GetMediaDebugLogEcho(ExtVol);
499 1 #endif

501 1      *External_volume = ExtVol;

503 1      return E_SUCCESS;
```



```
504 }          /* MediaObjectConstructor */
```

```
507  /*************************************************************
508   * validateMediaObject ()
509   *
510   *  This function updates the mediaObject, it updates the
511   *  fields that are likely to have changed since the volume
512   *  information first retrieved from rvm.
513   *
514   * Returns: void
515   *
516   * Parameters:
517   *   volid       (I) -  volume to be validated
518   *   thisObject (I/O) -  media object to be updated
519   *
520   *
521   *************************************************************/

523  static void
524  validateMediaObject(
                volumeID_ty *volid, struct RSTRPC_media_object *thisObject)
525  1 {
526  1 #define ERROR_BUF_SIZE 512
527  1     boolean_ty online;
528  1     boolean_ty offsite;
529  1     time_t lmtime;
530  1     char error_buf[ERROR_BUF_SIZE];
531  1     eerrno_ty ep_status = E_SUCCESS;

533  1     ep_status = ebvl_online_offsite_check( volid, &lmtime,
534  1                                            &online, &offsite,
535  1                                            error_buf, ERROR_BUF_SIZE);
536  1     if (E_SUCCESS != ep_status)
537  2     {
538  2          rbe_internal_error(0,
539  2                             "in call ebvl_online_offsite_check(): %s",
540  2                             error_buf);
541  1     }
542  1     else
543  2     {
544  2          thisObject->lmtime = lmtime;
545  2          thisObject->online = online;
546  2          thisObject->offsite = offsite;
547  1     }
548  1 #undef ERROR_BUF_SIZE
549  }         /* validateMediaObject */
```



```
551  /*************************************************************
552   * volid2str()
553   *
554   *  This function converts a volumeID_ty to a charactor string
555   *  that represents the volume in hexidecimal.
556   *  This function returns a char * that is a static and must
557   *  be copied prior to future calls to volid2str(). It is passed
558   *  a volumeID_ty as input.
559   *
560   * Returns:
561   *   static char * which must be copied.
562   * Parameters:
563   *   volid (I) -- input volumeID_ty
564   *
565   *************************************************************/

567  static char *
568  volid2str(const volumeID_ty *volid)
569  1 {
570  1     int index;
571  1     static char str_volid[20];


574  1     *str_volid = '\0';

576  1     if (volid == NULL)
577  2     {
578  2          return NULL;
579  1     }
580  1     sprintf(str_volid, "%08lX%08lX",
581  1             volid->opaque[0], volid->opaque[1]);
582  1     return str_volid;
583  }         /* volid2str */
```

```
585  /*****************************************************************
586   * AssignMediaObject():
587   *
588   *  This function is provided to assign MediaObjects.
589   *
590   * Returns:
591   *   status:
592   *       E_SUCCESS               if completed successfully
593   *       EP_RB_RECOVER_NOMEM     if memory allocation failed
594   *
595   * Parameters:
596   *   maxEntries     (I)- the maximum number of media objects to return
597   *   objects        (O)- pointer to linked list of media objects
598   *   Init_volume    (I)- head of internal volumes;
                                        to determine where to start
599   *   numberEntries (
              O)- the real number of media objects returned in the array
600   *   cookie          (
            O)- pointer to allow retrieval  of more objects if there is
601   *                       more than "maxEntries".
                                    The cookie will be returned as
602   *                       DONE_COOKIE when there are no more to
                                                        retrieve.
603   *****************************************************************/

605  static eerrno_ty
606  AssignMediaObjects( const ebvl_volidlist_ty      *Init_volume,
607                      const short                          maxEntries,
608                      struct RSTRPC_media_list     **objects,
609                      short                        *numberEntries,
610                      boolean_ty                   all,
611                      long                         *cookie )

613  1 {
614  1     register int MedObjCount;
615  1     long return_cookie = DONE_COOKIE;
616  1     register ebvl_volidlist_ty *Head_internal_list
617  1                                       = (
                                      ebvl_volidlist_ty *) Init_volume;
618  1     struct RSTRPC_media_object *temp_media;
619  1     struct RSTRPC_media_object *temp_Original_media;
620  1     struct RSTRPC_media_list *listPtr = NULL;
621  1     struct RSTRPC_media_list *tempList;
622  1     struct RSTRPC_media_list *t_tempList;
623  1     eerrno_ty result = E_SUCCESS;
624  1     ebvl_volidlist_ty  *temp_dups;
625  1     ebvl_volidlist_ty  *targetMediaEntry = NULL;
626  1     short           num_dups;
627  1     struct RSTRPC_media_list *dup_media_list;
628  1     char cVolIdDup[VOLUME_VOLID_LEN];
629  1     char cVolIdOri[VOLUME_VOLID_LEN];

631  1     /*
632  1      * Teminating conditions for loop are:
633  1      * 1) maxEntries mediaObjects have been loaded.
634  1      *
635  1      * OR
636  1      *
637  1      * 2) end of ebvl_volidlist encountered.
638  1      */

640  1     MedObjCount = 0;

642  1     while (MedObjCount < maxEntries)
643  2     {
```



```
644  2         /*
645  2          * If the end of the ebvl_volidlist is encountered
646  2          */

648  2         if (NULL == Head_internal_list)
649  3         {
650  3             return_cookie = DONE_COOKIE;
651  3             *numberEntries = (short) MedObjCount;
652  3             break;
653  2         }

655  2         if (DOES_VOLID_HAVE_MKD_FILES(Head_internal_list))
656  3         {
657  3             /*
658  3              * Only return mediaObject of volumes with marked
                                                                  bitfiles
659  3              */

661  3             /*
662  3              * Copy fields from ebvl_volidlist to mediaObject
663  3              */

665  3             if ( (E_SUCCESS !=
666  3                   (result = MediaObjectConstructor(
                                                  Head_internal_list,
667  3                                                 &temp_media,
                                                        FALSE )) ) )

668  4             {
669  5                 if (result != EP_RB_RECOVER_INVAL_OBJTYPE) {
670  5                         RSTSL_FreeMediaObjectList( *objects );
671  5                         *objects = NULL;
672  5                         return result;
673  4                 }

675  4                 /*
676  4                  *  if the mediaObjectConstructor failed,
677  4                  *  lets bail out and try the next ebvl_volidlist
678  4                  */

680  4                 Head_internal_list = Head_internal_list -> next;
681  4                 continue;
682  3             }

684  3             if (NULL == listPtr)
685  4             {
                                        /* first time thru, set list head */
686  4                 /* allocate new list entry,
                                                abort on malloc failure */
687  4                 listPtr = calloc( 1, sizeof(
                                              struct RSTRPC_media_list ));
688  4                 if(NULL == listPtr)
689  5                 {
690  5                         FreeMediaObjectContents( temp_media );
691  5                         RSTSL_FreeMediaObjectList( *objects );
692  5                         *objects = NULL;
693  5                         return EP_RB_RECOVER_NOMEM;
694  4                 }
695  4                 tempList  = listPtr;
696  4                 *objects = tempList;
697  3             }
698  3             else
699  4             {
700  4                 listPtr = calloc( 1, sizeof(
                                              struct RSTRPC_media_list ));
701  4                 if (NULL == listPtr )
702  5                 {
```

```
703  5                        FreeMediaObjectContents( temp_media );
704  5                        RSTSL_FreeMediaObjectList( *objects );
705  5                        *objects = NULL;
706  5                        return EP_RB_RECOVER_NOMEM;
707  4                    }
708  4                    for (t_tempList = tempList;
709  4                           ((NULL != t_tempList) && (
                                        NULL != t_tempList->next));
710  4                         t_tempList = t_tempList->next)
711  5                    {
712  5                           /* null */
713  4                    }
714  4                    if ( NULL == t_tempList )
715  5                    {
716  5                           FreeMediaObjectContents( temp_media );
717  5                           RSTSL_FreeMediaObjectList( *objects );
718  5                           *objects = NULL;
719  5                           return EP_RB_RECOVER_NOMEM;
720  4                    }

722  4                    t_tempList->next = listPtr;
723  3                }

725  3                /* Only return the preferred volume OR teh original
                                                              w/dup*/
726  3                if ((FALSE == all) || (TRUE == temp_media->is_orig))
727  4                 {
728  4                     listPtr->media_obj = temp_media;
729  3                 }
730  3                else if ((TRUE == all) && (FALSE == temp_media->is_orig))
731  4                 {
732  4                    /* Retrieve the original and assign to the return
                                                              list */
733  4                      if ((E_SUCCESS !=
734  4                     (result = MediaObjectConstructor(
                                              Head_internal_list,
735  4
                                             &temp_Original_media,
                                                TRUE ))))
736  4                      {
737  5                           if (result != EP_RB_RECOVER_INVAL_OBJTYPE)
738  5                             {
739  6                                   RSTSL_FreeMediaObjectList( *objects );
740  6                                    FreeMediaObjectContents(
741  6                                                          temp_media );
742  6                                   *objects = NULL;
743  6                                   return result;
744  5                             }
745  4                      }

747  4                      listPtr->media_obj = temp_Original_media;
748  4                    /* Assign the preferred duplicate to the return list
                                                                        */
749  4                      listPtr->media_obj->dups =
750  4                           calloc (1, sizeof(struct RSTRPC_media_list));
751  4                      if (NULL == listPtr->media_obj->dups)
752  5                      {
753  5                           RSTSL_FreeMediaObjectList( *objects );
754  5                           FreeMediaObjectContents( temp_media );
755  5                           FreeMediaObjectContents(
                                              temp_Original_media );
756  5                           *objects = NULL;
757  5                           return EP_RB_RECOVER_NOMEM;
758  4                      }
759  4                      listPtr->media_obj->num_dups = 1;
760  4                      listPtr->media_obj->dups->media_obj = temp_media;
```



```
762  3                 }


765  3                /*
766  3                 * Lets validate volume with rvm and update volatile
767  3                 * volume fields.
768  3                 *
769  3                * OSGsw31074 - The following code was modified to
                                                              validate
770  3                *      the volume in "temp_media" with the correct volume
771  3                *      in the "Head_internal_list". Before this fix was
772  3                *      inserted,
                                we always checked the against the original
773  3                *      volume. This had a side affect of resetting the
774  3                *      offline/online attributes in the duplicate volume.
775  3                 */

777  3                targetMediaEntry=Head_internal_list;
778  3                rvmvolm_volumeID_to_ascii(&Head_internal_list->volid,
779  3                                          cVolIdOri);
780  3                /* Is the volume in the temp_media the same as the
                                                              Original ? */
781  3                if ( 0 != (strcmp(cVolIdOri, temp_media->volid_ascii)) )
782  4                {
783  4                    /* Do we have a duplicate volume to check */
784  4                    if(NULL != Head_internal_list->dupsP)
785  5                     {
786  5                      /* Yes, then check the duplicate volume */
787  5                       rvmvolm_volumeID_to_ascii(
                                    &Head_internal_list->dupsP->volid,
788  5                                                 cVolIdDup);
789  5                       if ( 0 != (strcmp(
                                      cVolIdDup,temp_media->volid_ascii)) )
790  6                        {
791  6                            FreeMediaObjectContents( temp_media );
792  6                            FreeMediaObjectContents(
                                                   temp_Original_media );
793  6                            RSTSL_FreeMediaObjectList( *objects );
794  6                            *objects = NULL;
795  6                            return EP_RB_RECOVER_INVAL_OBJNAME;
796  5                        }
797  5                      /* Yes it is, so query/validate on the duplicate */
798  5                       targetMediaEntry=Head_internal_list->dupsP;
799  4                    }
800  4                   else
801  5                    {
802  5                           FreeMediaObjectContents( temp_media );
803  5                          RSTSL_FreeMediaObjectList( *objects );
804  5                           FreeMediaObjectContents(
                                                   temp_Original_media );
805  5                          *objects = NULL;
806  5                          return EP_RB_RECOVER_INVAL_OBJNAME;
807  4                    }
808  3                }
809  3                validateMediaObject(
                                &targetMediaEntry->volid, temp_media );



812  3                /*
813  3                 * handle Media dup logic here GOAL  (not any more)
814  3                 * Here is the Media Dup logic!!!!!!!!!!
815  3                 */

817  3                temp_dups = Head_internal_list->dupsP;
818  3                num_dups = 0;
```

```
820  3              if ( (TRUE == temp_media->is_orig) && (
                                        temp_media->run_media_dup) )
821  4                {
822  4                  temp_media->dups = calloc (1, sizeof(
                                        struct RSTRPC_media_list));
823  4                  dup_media_list = temp_media->dups;

825  4                  while (temp_dups != NULL)
826  5                    {


829  5                      if ( (E_SUCCESS !=
830  5                            (result = MediaObjectConstructor(temp_dups,
831  5
                                      &dup_media_list->media_obj, FALSE)) ) )
832  6                        {
833  7                          if (result != EP_RB_RECOVER_INVAL_OBJTYPE) {
834  7                            RSTSL_FreeMediaObjectList( *objects );
835  7                            *objects = NULL;
836  7                            return result;
837  6                          }

839  5                        }
840  5                      num_dups++;
841  5                      temp_dups=temp_dups->next;
842  5                      temp_media->num_dups = num_dups;

844  5                      if (temp_dups !=NULL)
845  6                        {
846  6                          dup_media_list->next = calloc (1, sizeof(
                                        struct RSTRPC_media_list));
847  6                          dup_media_list = dup_media_list->next;

849  5                        }
850  4                    }
851  3                }

853  3              if ((temp_media->run_media_dup) && (all == FALSE))
854  4                {
855  4                  if ((temp_media->num_dups>0) &&
856  4                      (temp_media->online == FALSE) &&
857  4                      (temp_media->dups->media_obj->online == TRUE))
858  5                    {
859  5                      listPtr->media_obj = temp_media->dups->media_obj;

860  5                      free(temp_media->dups);
861  5                      temp_media->dups = NULL;
862  5                      FreeMediaObjectContents(temp_media);
863  5                      free(temp_media);
864  5                      temp_media = NULL;
865  4                    }
866  4                  else
867  5                    {
868  5                      if ((temp_media->run_media_dup) && (
                                        temp_media->dups != NULL))
869  6                        {
870  6                          if (temp_media->dups->media_obj != NULL)
871  7                            {
872  7                              FreeMediaObjectContents(
                                        temp_media->dups->media_obj);
873  7                              free(temp_media->dups->media_obj);
874  6                            }
875  6                          free(temp_media->dups);
876  6                          temp_media->dups = NULL;
877  6                          temp_media->num_dups = 0;
```



```
878  5                        }

880  4                      }

882  3                    }

884  3                  MedObjCount++;
885  2                }
886  2                if (NULL != temp_media && NULL != temp_media->dups)
887  3                  {
888  3                    if (NULL == temp_media->dups->media_obj)
889  4                      {
890  4                        free(temp_media->dups);
891  4                        temp_media->dups = NULL;
892  3                      }
893  2                  }
894  2                /*
895  2                 * If we are going to terminate loop lets set up output
896  2                 * parameters and return cookie.
897  2                 */

899  2                if (maxEntries == MedObjCount)
900  3                {
901  3                    return_cookie = (long) Head_internal_list -> next;
902  3                    *numberEntries = (short) MedObjCount;
903  2                }

905  2                Head_internal_list = Head_internal_list -> next;
906  1            }

908  1            if (NULL == return_cookie)
909  2            {
910  2                /*
911  2                 * Correct for NULL pointer for the next
912  2                 * ebvl_volidlist which means DONE_COOKIE
913  2                 */

915  2                return_cookie = DONE_COOKIE;
916  1            }

918  1            /*
919  1             * The cookie will be a pointer to the next
920  1             * ebvl_volidlist or .. DONE_COOKIE.
921  1             */

923  1            *cookie = return_cookie;
924  1            return E_SUCCESS;
925       }        /* AssignMediaObjects */
```

```
927  /*
928   * debugging functions
929   */
930
931  /*****************************************************************
932   * GetMediaDebugLogEcho()
933   *
934   *  This function is provided to print mediaObjects to stdout or
935   *  log the mediaObject to recoveries.log.
936   *
937   * Returns:
938   *  void
939   *
940   * Parameters:
941   *  medObjPtr     (I)- the input mediaObject to print or log.
942   *
943   * Note:
944   *  if #define DEBUG is set then the mediaObject will be printed to
                                                              stdout.
945   *  if #define DEBUG_LOG_GET_NECC is set then the mediaObject will be
946   *          printed to recoveries.log.
947   *****************************************************************/
948
949  void
950  GetMediaDebugLogEcho(const mediaObject *MedObjPtr)
951  1 {
952  1 #if defined(DEBUG)
953
954  1     printf("\tVOLIDascii=%s\n", MedObjPtr -> volid_ascii);
955
956  1     printf("Trail = %s\t", (NULL == MedObjPtr -> trail) ? "no trail":
957  1             MedObjPtr -> trail);
958
959  1     printf("Mtype = %s\t ", (NULL == MedObjPtr -> mtype) ? "no mtype":
960  1             MedObjPtr -> mtype);
961
962  1     printf("Barcode = %s\n ", (
                      NULL == MedObjPtr -> barcode_label) ? "no barcode":
963  1             MedObjPtr -> barcode_label);
964
965  1     printf("Physical location = %s\n ", (
             NULL == MedObjPtr -> physical_loc) ? "no physical location":
966  1             MedObjPtr -> physical_loc);
967
968  1     printf("Comments = %s\n ", (
                    NULL == MedObjPtr -> comments) ? "no user comments":
969  1             MedObjPtr -> comments);
970
971  1     printf("Sequence Number = %d, Side = %c\t", MedObjPtr -> seqno,
972  1             (MedObjPtr -> side == '\0') ? ' ' :MedObjPtr -> side);
973
974  1     printf("Online = %s, Offsite = %s\n",
975  1             (FALSE == MedObjPtr -> online)?"FALSE":"TRUE",
976  1             (FALSE == MedObjPtr -> offsite)?"FALSE":"TRUE");
977
978  1     printf("Is_orig = %s, running_media_dup = %s\n",
979  1             (FALSE == MedObjPtr -> is_orig)?"FALSE":"TRUE",
980  1             (FALSE == MedObjPtr -> run_media_dup)?"FALSE":"TRUE");
981
982
983  1 #endif
984  1 #if defined(DEBUG_LOG_GET_NECC)
985
986  1     rbe_log_stats(0, "VOLID = %s Trail = %s",
987  1                   MedObjPtr -> volid_ascii,
```



```
988  1                    (
           NULL == MedObjPtr -> trail) ? "no trail" : MedObjPtr -> trail);
989
990  1     rbe_log_stats(0, "Barcode = %s", (
                                        NULL == MedObjPtr -> barcode_label)
991  1                   ? "no barcode" : MedObjPtr -> barcode_label);
992
993  1     rbe_log_stats(0, "Physical location = %s",
994  1                   (NULL == MedObjPtr -> physical_loc)
995  1                   ? "no physical location" :  MedObjPtr ->
                                                          physical_loc);
996
997  1     rbe_log_stats(0, "Comments = %s",
998  1                   (NULL == MedObjPtr -> comments)
999  1                   ? "no user comments" : MedObjPtr -> comments);
1000
1001 1     rbe_log_stats(0, "Mtype = %s Sequence Number = %d, Side = %c ",
1002 1                   (NULL == MedObjPtr -> mtype)
1003 1                   ? "no mtype" : MedObjPtr -> mtype,
1004 1                   MedObjPtr -> seqno,
1005 1                   (
                 MedObjPtr -> side == '\0') ? ' ' : MedObjPtr -> side);
1006
1007 1     rbe_log_stats(0, "Online = %s, Offsite = %s",
1008 1                   (FALSE == MedObjPtr -> online) ? "FALSE" : "TRUE",
1009 1                   (FALSE == MedObjPtr -> offsite) ? "FALSE" : "TRUE");
1010
1011 1     rbe_log_stats(0, "Is_orig = %s, Run_media_dup = %s",
1012 1                   (FALSE == MedObjPtr -> is_orig) ? "FALSE" : "TRUE",
1013 1                   (
              FALSE == MedObjPtr -> run_media_dup) ? "FALSE"  : "TRUE");
1014 1 #endif
1015  }         /* GetMediaDebugLogEcho */
```

```
1018   /*****************************************************************
1019    * loadMediaObject()
1020    *
1021    *  This function is a routine to load a mediaObject with garbage
1022    *  fields, It was useful during debugging and may prove useful in
1023    *  the future.
1024    *
1025    * Returns:
1026    *  eerrno_ty
1027    *
1028    * Parameters:
1029    *  medObjPtr     (O)- the output mediaObject to be loaded.
1030    *
1031    *****************************************************************/

1033   eerrno_ty
1034   loadMediaObject(struct RSTRPC_media_object *MedObjPtr)
1035 1 {
1036 1     volumeID_ty  volid;

1038 1     MedObjPtr-> trail = esl_strdup("backup_DLT");
1039 1     if (NULL == MedObjPtr->trail)
1040 2     {
1041 2          rec_api_log_csm(SUB_CSM_NOMEM, NULL);
1042 2          return(EP_RB_RECOVER_NOMEM);
1043 1     }

1045 1     MedObjPtr-> mtype = esl_strdup("DLT");
1046 1     if (NULL == MedObjPtr->mtype)
1047 2     {
1048 2          rec_api_log_csm(SUB_CSM_NOMEM, NULL);
1049 2          return(EP_RB_RECOVER_NOMEM);
1050 1     }

1052 1     MedObjPtr-> barcode_label = esl_strdup("134545325BARCODE");
1053 1     if (NULL == MedObjPtr->barcode_label)
1054 2     {
1055 2          rec_api_log_csm(SUB_CSM_NOMEM, NULL);
1056 2          return(EP_RB_RECOVER_NOMEM);
1057 1     }

1059 1     MedObjPtr-> physical_loc = esl_strdup("Under the table you idiot");
1060 1     if (NULL == MedObjPtr->physical_loc)
1061 2     {
1062 2          rec_api_log_csm(SUB_CSM_NOMEM, NULL);
1063 2          return(EP_RB_RECOVER_NOMEM);
1064 1     }

1066 1     MedObjPtr-> comments = esl_strdup("The is my favorite volume");
1067 1     if (NULL == MedObjPtr->comments)
1068 2     {
1069 2          rec_api_log_csm(SUB_CSM_NOMEM, NULL);
1070 2          return(EP_RB_RECOVER_NOMEM);
1071 1     }

1073 1     ((long *) &volid)[0] = -5146245;
1074 1     ((long *) &volid)[1] = -3456776;
1075 1     MedObjPtr->volid_ascii = esl_strdup(volid2str(&volid));
1076 1     if (NULL == MedObjPtr->volid_ascii)
1077 2     {
1078 2          rec_api_log_csm(SUB_CSM_NOMEM, NULL);
1079 2          return(EP_RB_RECOVER_NOMEM);
1080 1     }
```



```
1082 1     MedObjPtr->seqno = 123743;
1083 1     MedObjPtr->side = 'a';
1084 1     MedObjPtr->lmtime = 0x31657905;
1085 1     MedObjPtr->online = TRUE;
1086 1     MedObjPtr->offsite = FALSE;
1087 1     MedObjPtr->is_orig = TRUE;
1088 1     MedObjPtr->run_media_dup = FALSE;
1089   }         /* loadMediaObject */
```

D9

Wed Oct 08 16:41:14 2008 Function Index Page 2

Wed Oct 08 16:41:14 2008 Function Index Page 1

Wed Oct 08 16:41:14 2008 Function Index Page 2

```
/*****************************************************************************
                                                                           *****
**
** File Name:   RSTstart.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
** -----------
**       The intent of the contents of this file is to implement the
**       functions the control execution of the restore for the
                                                              Restore API.
**
**       These functions are provided to allow:
**      - creation of submit objects,
                                  which define the set of objects to be
**        restored and the scripts to be run before and after
                                                              restoration,
**      - starting the restoral of a submit object.
**      - polling of the status of an ongoing restore,
                                                 including the ability to
**        interrupt the restore,
                                  and to receive information necessary to
**        query the user for input needed for the pre-restore or
                                                              post-restore
**        scripts, suspending, restarting,
**      - supply of user responses to pre- and post- restore script
                                                                  queries
**
** The following functions comprise restoral management:
**
**      EDMRST_Submit
**      EDMRST_GetSubmitResults
**      EDMRST_Start
**      EDMRST_GetRestoreFeedback
**      EDMRST_GetQuestion
**      EDMRST_SetUserAnswer
**
**
** Compile-Time Options:
**              This section must list any compile time definitions
**              which will affect this header.
**
*****************************************************************************
                                                                       *****/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */
```



```
#define  _POSIX_SOURCE 1
#define  NULL_STRING "\0"

/*
 * System headers.
 */

#include <sys/wait.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/eb_normalize.h>
#include <ebutil/ebutil.h>
#include <ebreport/ebv1.h>


/*
 * Local headers
 */
#include <RSTinterns.h>
#include <RSTsup_csm.h>



/*
 * #defines, structures, typedefs local to this source file
 */



/*
 * Command flags.
 */


/*
 * External declarations
 */


/*
 * Local function prototypes
 */
```

```
/*****************************************************************
 * Restoral Management Functions:
 *
 *  These functions are provided to allow:
 *      - creation of submit objects, which define the set of objects to be
 *        restored and the scripts to be run before and after restoration,
 *      - starting the restoral of a submit object.
 *      - polling of the status of an ongoing restore, including the ability to
 *        interrupt the restore, and to receive information necessary to
 *        query the user for input needed for the pre-restore or post-restore
 *        scripts, suspending, restarting,
 *      - supply of user responses to pre- and post- restore script queries
 *
 * The following functions comprise restoral management:
 *
 *      EDMRST_Submit
 *      EDMRST_GetSubmitResults
 *      EDMRST_Start
 *      EDMRST_GetRestoreFeedback
 *       EDMRST_GetQuestion
 *      EDMRST_SetUserAnswer
 *      EDMRST_SetRecxDirectives
 *
 *****************************************************************
 * Submit
 *
 * This function starts the creation or update of a submit object from the
 * currently marked restorable objects. Its completion is tested for with
 * EDMRST_GetSubmitResults. The returned submit object ID is passed to
 * EDMRST_Start to begin execution of the restore.
 *
 * Parameters:
 *
 * svrHdl        (I) - A pointer to this user's client handle for
 *                     the Restore Engine (server) connection.
 * policy        (I) - The overwrite policy to use
 * inPlace       (I) - Flag if the restoral is to be in original locations
 * hostName      (I) - host to restore to (only if inPlace == False)
 * directory     (I) - directory to restore to (only if inPlace == False)
 * transport     (I) - Indicator of transport the restoral is to be over (SCSI
 *                     or network)
 * submitObjID   (I) - ID of an existing submit object which is to be added to
 * socketClientNm (I) - Name of the client the restore is going to on
 *                      a client initiated restore
 * clientSocketPort (I) - the port to connect to on the client machine for
 * the restore
 *****************************************************************/
eerrno_ty EDMRST_Submit( serverHandle          svrHdl,
                         const char            *hostName,
                         const OverwritePolicy policy,
                         const boolean_ty      inPlace,
                         const char            *directory,
```



```
                         const RestoreTransport transport,
                         unsigned int          submitObjID,
                         EDMRST_submit_args    *submitArgs)
{
        RE_status_result        *rpc_result;
        RE_submit_args          rpc_args;
        eerrno_ty               result;
        char                    *nullstr = "";

        /* validate args first: */
        if ( svrHdl == NULL
          || (NULL == handlePtr) || (
                                svrHdl != handlePtr->re_binding_handle)
          || (!inPlace && (hostName == NULL || directory == NULL) )
        )
                return( EP_RB_RECOVER_BAD_ARGS );

        rpc_args.overwritePolicy = policy;
        rpc_args.inPlace = inPlace;
        rpc_args.transport = transport;
        rpc_args.submitObjectID = submitObjID;

        if (NULL != submitArgs)
          {
            rpc_args.mapFile_env = esl_strdup(
                                                submitArgs->mapfile_env);
          }
        else
          {
            rpc_args.mapFile_env = nullstr;
          }

        if (NULL != submitArgs)
          {
            rpc_args.socketPort=submitArgs->clientSocketPort;
          }
        else
          {
            rpc_args.socketPort = 0;
          }

        if (NULL != submitArgs)
          {
            if (NULL==(rpc_args.socketClientName=esl_strdup(
                                                submitArgs->socketClientNm)))
              rpc_args.socketClientName=nullstr ;
          }
        else
          {
            rpc_args.socketClientName=nullstr ;
          }

        if (!inPlace) {
                rpc_args.hostname = (char *)hostName;
                rpc_args.directory = (char *)directory;
        } else {
                rpc_args.hostname = nullstr;
                rpc_args.directory = nullstr;
        }

        set_rpc_obj( re_submit, &rpc_args.RPCobjID );

        rpc_result = re_submit_1( &rpc_args, svrHdl );

        if (!rpc_result) {
                result = EP_RB_RECOVER_RPC_FAIL;
```

```
222  2                  rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
223  1          }
224  1          else
225  2          {
226  2                  result = rpc_result->status;

228  2                  /* release RPC result struct: contents and struct */
229  2                  xdr_free( xdr_RE_status_result, (char *)rpc_result );
230  1          }
231  1          return( result );
232     }
```



```
235     /*********************************************************************
236      * GetSubmitResults
237      *
238      * This function tests for completion of an EDMRST_Submit call,
                                                                    with the
239      * option of cancelling the submit.
240      *
241      * Parameters:
242      *
243      * svrHdl        (I) - A pointer to this user's client handle for
244      *                     the Restore Engine (server) connection.
245      * interrupt     (I) - Flag if the submit is to be canceled
246      * submitObjID   (
                 O) - ID of the submit object which describes the restore
247      * objectsDone   (
            O) - number of objects -- total number in the submit object
248      *                      if operation is complete,
                                        or number processed so far if
249      *                      submit operation is still executing (
                                              INCOMPLETE status)
250      *
251      *********************************************************************/
252     eerrno_ty EDMRST_GetSubmitResults( serverHandle     svrHdl,
253                                        const boolean_ty interrupt,
254                                        unsigned int     *submitObjID,
255                                        unsigned long    *objectsDone )
256  1  {
257  1          RE_get_submit_results_output    *rpc_result;
258  1          RE_get_submit_results_args      rpc_args;
259  1          eerrno_ty                       result;

261  1          /* validate args first: */
262  1          if (
                 svrHdl == NULL || submitObjID == NULL || objectsDone == NULL
263  1          || (NULL == handlePtr) || (
                                    svrHdl != handlePtr->re_binding_handle)
264  1          )
265  1                  return( EP_RB_RECOVER_BAD_ARGS );

267  1          rpc_args.interrupt = interrupt;

269  1          set_rpc_obj( re_get_submit_results, &rpc_args.RPCobjID );

271  1          rpc_result = re_get_submit_results_1( &rpc_args, svrHdl );

273  2          if (!rpc_result) {
274  2                  result = EP_RB_RECOVER_RPC_FAIL;
275  2                  rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
276  1          }
277  1          else
278  2          {
279  2                  result = rpc_result->status;
280  2                  *objectsDone = rpc_result-> objectsDone;
281  2                  if (result == E_SUCCESS)
282  2                          *submitObjID = rpc_result->submitObjectID;

284  2                  /* release RPC result struct: contents and struct */
285  2                  xdr_free (xdr_RE_get_submit_results_output, (
                                                        char *)rpc_result);
286  1          }
287  1          return( result );
288     }
```

```
290   /*****************************************************************
291    * Start
292    *
293    * This function begins execution of the restoral of the objects in a
294    * submit object.  Its progress and requests for operator input are
295    * received via EDMRST_GetRestoreFeedback.
296    *
297    * Parameters:
298    *
299    * svrHdl        (I) - A pointer to this user's client handle for
300    *                     the Restore Engine (server) connection.
301    * submitObjID   (
                    I) - ID of the submit object that describes the restore
302    *
303    *****************************************************************/
304   eerrno_ty EDMRST_Start( serverHandle  svrHdl,
305                           unsigned int submitObjID )
306 1 {
307 1         RE_status_result        *rpc_result;
308 1         RE_start_args           rpc_args;
309 1         eerrno_ty               result;

311 1         /* validate args first: */
312 1         if (svrHdl == NULL || submitObjID == 0
313 1         || (NULL == handlePtr) || (
                                  svrHdl != handlePtr->re_binding_handle)
314 1         )
315 1                 return( EP_RB_RECOVER_BAD_ARGS );

317 1         rpc_args.submitObjectID = submitObjID;

319 1         set_rpc_obj( re_start, &rpc_args.RPCobjID );

321 1         rpc_result = re_start_1( &rpc_args, svrHdl );

323 2         if (!rpc_result) {
324 2                 result = EP_RB_RECOVER_RPC_FAIL;
325 2                 rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
326 1         }
327 1         else
328 2         {
329 2                 result = rpc_result->status;

331 2                 /* release RPC result struct: contents and struct */
332 2                 xdr_free( xdr_RE_status_result, (char *)rpc_result );
333 1         }
334 1         return( result );
335   }       /* EDMRST_Start */
```



```
336   /*****************************************************************
337    * GetRestoreFeedback
338    *
339    * This function is used to poll for the status of an ongoing restore,
                                                                    and
340    * includes the ability to interrupt the restore,
                                                  and to receive information
341    * necessary to query the user for input needed for the pre-restore or
342    * post-restore scripts.
343    *
344    * Parameters:
345    *
346    * svrHdl        (I) - A pointer to this user's client handle for
347    *                     the Restore Engine (server) connection.
348    * quitRestore   (I) - Flag if the restoral is to be stopped
349    * currentState  (
                O) - Pointer to storage to receive the state of the restore
350    * feedbackPtr   (
                IO) - Pointer to structure to receive restore feedback data
351    *
352    *****************************************************************/

354   eerrno_ty EDMRST_GetRestoreFeedback( serverHandle     svrHdl,
355                                        const boolean_ty quitRestore,
356                                        RERunningState   *currentState,
357                                        feedbackObjectPtr feedbackPtr )
358 1 {
359 1         RE_get_restore_feedback_result  *rpc_result;
360 1         RE_get_restore_feedback_args    rpc_args;
361 1         feedbackObject  *foPtr = (feedbackObject *)feedbackPtr;
362 1         eerrno_ty               result;

364 1         /* validate args first: */
365 1         if (
          svrHdl == NULL || currentState == NULL || feedbackPtr == NULL
366 1         || (NULL == handlePtr) || (
                                      svrHdl != handlePtr->re_binding_handle)
367 1          || FEEDBACK_OBJECT != foPtr->restoreObjType
368 1         )
369 1                 return( EP_RB_RECOVER_BAD_ARGS );

371 1         FreeFeedbackObjectContents( foPtr );

373 1         rpc_args.quit_restore = quitRestore;

375 1         set_rpc_obj( re_get_restore_feedback, &rpc_args.RPCobjID );

377 1         rpc_result = re_get_restore_feedback_1( &rpc_args, svrHdl );

379 2         if (!rpc_result) {
380 2                 result = EP_RB_RECOVER_RPC_FAIL;
381 2                 rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
382 1         }
383 1         else
384 2         {
385 2                 result = rpc_result->status;
386 2                 foPtr->stats.status = rpc_result->rstStats.status;
387 2                 foPtr->stats.wiprogress = rpc_result->rstStats.
                                                          wiprogress;
388 2                 rpc_result->rstStats.wiprogress = NULL;
                                                  /* avoid 2 frees */
389 2                 memcpy( &foPtr->stats.edm, &rpc_result->rstStats.edm,
390 2                         sizeof (struct EDMProgress) );
391 2                 rpc_result->rstStats.edm.next = NULL;
                                                  /* for xdr_free */
392 2                 foPtr->notify = rpc_result->notify;
```

```
393 2                   rpc_result->notify = NULL;      /* avoid 2 frees */
394 2                   *currentState = rpc_result->rstStats.edm.status;

396 2                   /* release RPC result struct: contents and struct */
397 2                   xdr_free( xdr_RE_get_restore_feedback_result,
398 2                             (char *)rpc_result );
399 1           }

401 1           return( result );
402   }       /* EDMRST_GetRestoreFeedback */
```



```
404   /*****************************************************************
405    * SetUserAnswer
406    *
407    * This function is used to return user input requested via the
                                                              queryObjectPtr
408    * parameter output of the EDMRST_GetQuestion function call.
409    *
410    * Parameters:
411    *
412    * svrHdl   (I) - A pointer to this user's client handle for
413    *                the Restore Engine (server) connection.
414    * queryPtr (I) - Pointer to object containing the question data.
415    * answer   (I) - pointer to text string response to question
416    * more     (
           I) - indicator that there will be more answers to this question
417    *
418    *****************************************************************/

420   eerrno_ty EDMRST_SetUserAnswer( serverHandle     svrHdl,
421                                   queryObjectPtr   queryPtr,
422                                    const char      *answer,
423                                    boolean_ty      more )
424 1 {
425 1           RE_status_result        *rpc_result;
426 1           RE_set_user_answer_args         rpc_args;
427 1           Answer                  *answerObj, *tmpAnswer;
428 1           queryObject             *queryObj = (queryObject *)queryPtr;
429 1           eerrno_ty               result = E_SUCCESS;

431 1           /* validate args first: */
432 1           if (svrHdl == NULL || answer == NULL || queryPtr == NULL
433 1           || (NULL == handlePtr) || (
                                       svrHdl != handlePtr->re_binding_handle)
434 1            || (QUERY_OBJECT != queryObj->restoreObjType)
435 1            || (NULL == queryObj->query)
436 1           )
437 1                   return EP_RB_RECOVER_BAD_ARGS;

439 1           /* allocate answer list if none in queryObject yet: */
440 1           if (NULL == queryObj->answers)
441 2           {
442 2               if (NULL == (queryObj->answers = calloc( 1, sizeof(
                                                              AnswerList) )) )
443 2                   return EP_RB_RECOVER_NOMEM;
444 1           }

446 1           /* allocate and initialize answer object */
447 1           if (NULL == (answerObj = calloc( 1, sizeof(Answer) )))
448 1                   return EP_RB_RECOVER_NOMEM;
449 1           if (NULL == (answerObj->ctext = esl_strdup( answer )))
450 2           {
451 2                   free( answerObj );
452 2                   return EP_RB_RECOVER_NOMEM;
453 1           }
454 1           answerObj->qnum = queryObj->query->qnum;
455 1           answerObj->nextanswer = NULL;

457 1           ++queryObj->answers->numanswers;
458 1           if (NULL == (tmpAnswer = queryObj->answers->firstanswer))
459 1                   queryObj->answers->firstanswer = answerObj;
460 1           else
461 2           {
462 2                   while (NULL != tmpAnswer->nextanswer)
463 2                           tmpAnswer = tmpAnswer->nextanswer;
464 2                   tmpAnswer->nextanswer = answerObj;
```

```
465 1           }
467 1           if (more)
468 1                   return result;
470 1           /* prepare arg structures:  move answer list to rpc structure
                                                                        */
472 1           rpc_args.answers.numanswers = queryObj->answers->numanswers;
473 1           rpc_args.answers.firstanswer = queryObj->answers->firstanswer;
474 1           queryObj->answers->firstanswer = NULL;
475 1           set_rpc_obj( re_set_user_answer, &rpc_args.RPCobjID );
477 1           rpc_result = re_set_user_answer_1( &rpc_args, svrHdl );
479 2           if (!rpc_result) {
480 2                   result = EP_RB_RECOVER_RPC_FAIL;
481 2                   rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
482 1           }
483 1           else
484 2           {
485 2                   result = rpc_result->status;
487 2                   /* release RPC result struct: contents and struct */
488 2                   xdr_free( xdr_RE_status_result, (char *)rpc_result );
489 1           }
491 1           return( result );
492     }
```



```
496     /*****************************************************************
497      * GetQuestion
498      *
499      * This function is used to fetch the data needed to query the user
                                                                    during a
500      * pre-restore or post-restore script execution.
501      *
502      * Parameters:
503      *
504      * svrHdl    (I) - A pointer to this user's client handle for
505      *                 the Restore Engine (server) connection.
506      * queryPtr (O) - Pointer to the object containing the question data.
507      *
508      *****************************************************************/
510     eerrno_ty EDMRST_GetQuestion( serverHandle       svrHdl,
511                                   queryObjectPtr     queryPtr )
512 1   {
513 1           RE_get_question_result  *rpc_result;
514 1           queryObject             *query_ptr = (queryObject *)queryPtr;
515 1           RE_null_args            rpc_args;
516 1           eerrno_ty               result;
518 1           /* validate args first: */
519 1           if (svrHdl == NULL || queryPtr == NULL
520 1           || (NULL == handlePtr) || (
                                        svrHdl != handlePtr->re_binding_handle)
521 1            || (QUERY_OBJECT != query_ptr->restoreObjType )
522 1           )
523 1                   return( EP_RB_RECOVER_BAD_ARGS );
525 1           /* free last question in query obj */
526 1           FreeQueryObjectContents( query_ptr );
528 1           set_rpc_obj( re_get_question, &rpc_args.RPCobjID );
530 1           rpc_result = re_get_question_1( &rpc_args, svrHdl );
532 2           if (!rpc_result) {
533 2                   result = EP_RB_RECOVER_RPC_FAIL;
534 2                   rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
535 1           }
536 1           else
537 2           {
538 2                   result = rpc_result->status;
539 2                   query_ptr->query = rpc_result->query;
                                                        /* use returned obj */
540 2                   rpc_result->query = NULL;       /* avoid 2 frees */
542 2                   /* release RPC result struct: contents and struct */
543 2                   xdr_free( xdr_RE_get_question_result, (
                                                      char *)rpc_result );
544 1           }
546 1           return( result );
547     }
```

```
548          /* EDMRST_GetQuestion */

550  /*****************************************************************
551   * SetRecxDirectives:
552   *
553   *  This routine returns sends the Filename and path plus hostname
554   *    of the recx directives file, which was created by the command
555   *    eb_dc_restore, to the server which then processes the recx
556   *     directives
557   *
558   * Parameters:
559   *  svrHdl    (I) - A pointer to this user's client handle for the
560   *                  Restore Engine (server) connection.
561   *  template  (O) - The name of the local recx file
562   *  alternate (O) - the name of this host so the file can be tranfered
563   *
564   *****************************************************************/
565  eerrno_ty EDMRST_SetRecxDirectives( serverHandle     svrHdl,
566                                      char             *filename,
567                                     char             *hostname )
568  1 {

570  1          RE_status_result         *rpc_result;
571  1          RE_recx_file_info        rpc_args;
572  1          RSTRPC_recx_file_info    fileinfo;
573  1          eerrno_ty                result;
574  1          char                     *nullstr = "";
575  1          RE_status_result         *poll_result;
576  1          RE_null_args             args;
577  1          int                      count;

579  1          /* validate args first: */
580  1          if ( svrHdl == NULL
581  1           || (NULL == handlePtr) || (
                                  svrHdl != handlePtr->re_binding_handle)
582  1            || (hostname == NULL || filename == NULL || 0==strcmp(
                               hostname,"")|| 0==strcmp(filename,"")) )
583  1                 return( EP_RB_RECOVER_BAD_ARGS );

585  1          fileinfo.filename = esl_strdup(filename);
586  1          fileinfo.hostname = esl_strdup(hostname);
587  1          rpc_args.fileinfo = fileinfo;

589  1          /*no object ID in file info structure*/
590  1          set_rpc_obj( re_load_recx_directives, &rpc_args.RPCobjID );

592  1          rpc_result = re_load_recx_directives_1( &rpc_args, svrHdl );

594  2          if ((NULL==rpc_result) || (
                                      rpc_result->status != E_SUCCESS)) {
595  2                 result = EP_RB_RECOVER_RPC_FAIL;
596  2                 rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
597  1          }
598  1          else
599  2          {
600  2                 set_rpc_obj(
                          re_poll_load_recx_directives, &rpc_args.RPCobjID );

602  2                 /* Initialize the count and poll_result*/
603  2                 poll_result =  re_poll_load_recx_directives_1(
                                                      &args, svrHdl );
604  2                 count = 0;
605  2                 /* check to see if the RPC is still running
606  2                    until it finishes,
                                      or it is longer than 60 seconds.*/
607  2                 while((
```



```
     poll_result->status == EP_RB_RECOVER_RPC_INCOMPLETE)&&(count <=60))
608  3                    {
609  3                      poll_result =  re_poll_load_recx_directives_1(
                                                        &args, svrHdl );
610  3                      sleep(1);
611  3                      count++;
612  2                    }

614  2                  if (poll_result->status != E_SUCCESS)
615  2                    result = EP_RB_RECOVER_RPC_FAIL;
616  2                  else
617  2                    result = rpc_result->status;

619  2                  /* release RPC result struct: contents and struct */
620  2                  xdr_free( xdr_RE_status_result, (char *)rpc_result );
621  1          }

623  1          /*polling info stuf*/




627  1          return( result );
628     }
```

```
631  /*****************************************************************
632   * EDMRST_get_catalog_info:
633   *
634   *  This routine returns sends the fills the level string with the
635   *  level for backup being restored
636   *
637   * Parameters:
638   *  svrHdl           (
                       I) - A pointer to this user's client handle for the
639   *                        Restore Engine (server) connection.
640   *  backup_time      (I) - Time of the backup that is beeing looked at
641   *  *level           (O) - The level of the backup for specified time
642   *                         taken from catalog structure.  If not enough
643   *                         memory has been allocated value will be "\0"
644   *  *numrec          (
                        O) - The number of records for the specified backup
645   *                         taken from catalog structure.   If not enough
646   *                         memory has been allocated value will be "\0"
647   *  *catType         (O) - The type of catalog for the specified bacup
648   *                         taken from catalog structure.   If not enough
649   *                         memory has been allocated value will be "\0"
650   * Return Codes:
651   *   EP_RB_RECOVER_BAD_ARGS - arguments passed in are null
652   *   E_SUCCESS              - the fields have been filled in
653   *                            and RPC succeeded
654   *
655   *****************************************************************/
656  eerrno_ty
657  EDMRST_getCatalogInfo( serverHandle   svrHdl,
658                         time_t          backup_time,
659                         char            *level,
660                         char            *numrec,
661                         char            *catType)
662  1 {

664  1          RE_catalog_info                 *rpc_result;
665  1          RE_time                 rpc_args;
666  1          char                    *result = NULL;
667  1          int                     tmp;
668  1          /* validate args first: */
669  1          if ((0==backup_time) || (NULL==svrHdl ) || (NULL==level)
670  1              || (NULL==numrec) || (NULL==catType))
671  1                  return(EP_RB_RECOVER_BAD_ARGS);

673  1          /* Prepare input argument structure for RPC: */
674  1          rpc_args.backupTime=backup_time;
675  1          set_rpc_obj( re_get_catalog_info, &rpc_args.RPCobjID );

677  1          rpc_result = re_get_catalog_info_1( &rpc_args, svrHdl );

679  1          if (rpc_result == NULL)
680  2            {
681  2              rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
682  2              return(EP_RB_RECOVER_RPC_FAIL);
683  1            }

685  1         /*copy the structures level field to the level variable*/

687  1          tmp = rpc_result->level[0];
688  1          sprintf(level, "%d",tmp);
689  1         /*copy the structures numrec  field to the numrec variable*/
690  1         numrec=strcpy(numrec,rpc_result->numrec);
691  1         /*copy the structures catType field to the catType variable*/
692  1         catType=strcpy(catType,rpc_result->catType);
```



```
694  1          return( E_SUCCESS );

696  }
```

Fri Oct 10 16:16:06 2008 File Index Page 2

Fri Oct 10 16:16:06 2008 File Index Page 1

Fri Oct 10 16:16:06 2008 File Index Page 2

```
/*****************************************************************
                                                          ******
**
** File Name:   RSLsubmit.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
** -----------
**      The intent of the contents of this file is to implement the
**      functions to create the submitObject and submitElement.
**
**      These functions are provided to allow:
**      - creation of submit objects,
                                which define the set of objects to be
**        restored and the scripts to be run before and after
                                                        restoration,
**
** The following functions comprise restoral management:
**
**      RSTSL_Submit
**
**
** Compile-Time Options:
**              This section must list any compile time definitions
**              which will affect this header.
**
*****************************************************************
                                                          *****/



/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1
#define ULONG_TO_CHAR_SIZE 16


/*
 * System headers.
 */
/* for CreateSubmitname */
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
/* for CreateSubmitname */
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/eb_normalize.h>
#include <ebutil/ebutil.h>
#include <ebreport/ebvl.h>
#include <restore/RSLplugin.h>
```



```
/*
 * Local headers
 */

#include <RSLinterns.h>
#include <restore/EDMRESubmitApi.h>




void
fill_client_dirtop(struct recover_context *rcx);

static int
push_submit_file(struct recover_context *rcx,
                 int fd,
                 struct mark_summary *this_submit_files,
                 ebvl_volidlist_ty **this_submit_volumes,
                 struct mark_summary *total_submit_files,
                 ebvl_volidlist_ty **total_submit_volumes,
                 RSTSL_SubmitProgressProc progressCB,
                 boolean_ty *submitCancelled);


static int
push_bfinfo_to_submitfile(struct recover_context *rcx,
                          int plane,
                          cat_descriptor *catd,
                          long lmno,
                          int fd,
                          ebfs_uid_ty *prev_ebd,
                          struct mark_summary *this_submit_files,
                          ebvl_volidlist_ty **this_submit_volumes,
                          struct mark_summary *total_submit_files,
                          ebvl_volidlist_ty **total_submit_volumes);

static int
push_to_submitfile(int fd,
                   char *buf,
                   uint_t nbytes);

static void
ebfsid2str_lz(ebfs_uid_ty *ebfsidp,
              register char *buf);

static int
ssID2ebfd(rbssID_t *ssidp,
          ebfs_uid_ty *ebfdp);




/*************************************************************************
 * Restoral Management Functions:
 *
 *  These functions are provided to allow:
 *      - creation of submit objects,
                                  which define the set of objects to be
 *        restored and the scripts to be run before and after
                                                        restoration,
 *      - starting the restoral of a submit object.
 *
 * The following functions comprise restoral management:
 *
 *      RSTSL_Submit
 *      RSTSL_Start
```

```
127     *
128     ****************************************************************
129     * Submit
130     *
131     * This function creates a submit object from the currently marked
132     * restorable objects. It is passed to RSTSL_Start to begin execution
133     * of the restore.
134     *
135     * Parameters:
136     *
137     * policy        (I) - The overwrite policy to use
138     * inPlace       (
                  I) - Flag if the restoral is to be in original locations
139     * hostName      (I) - host to restore to (only if inPlace == False)
140     * directory     (I) - directory to restore to (
                                              only if inPlace == False)
141     * transport      (
              I) - Indicator of transport the restoral is to be over (SCSI
142     *                    or network)
143     * submitObjIDptr(
                   IO) - ID of the submit user object created to describe
144     *                    the restore
145     * ObjectsSubmitted (O) - number of total file objects submitted.
146     * progressCB    (
                  I) - pointer to callback function to report progress and
147     *                    test for cancellation
148     *
149     ****************************************************************/
150     eerrno_ty RSTSL_Submit(const char              *hostName,
151                            const OverwritePolicy   policy,
152                            const boolean_ty        inPlace,
153                            const char              *directory,
154                            const RestoreTransport  transport,
155                            int                      *submitObjID,
156                             unsigned int            *ObjectsSubmitted,
157                             RSTSL_SubmitProgressProc progressCB,
158                             EDMRST_submit_args       *submitArgs)
159   1 {

161   1   int submitElemID;
162   1   int ret_status;
163   1   int SEstatus = E_SUCCESS;
164   1   int SOstatus = E_SUCCESS;
165   1   int submit_fd;
166   1   char submit_filename[2048];
167   1   struct mark_summary *this_submit_files;
168   1   ebvl_volidlist_ty *this_submit_volumes = NULL;
169   1   struct mark_summary *total_submit_files;
170   1   ebvl_volidlist_ty *total_submit_volumes = NULL;
171   1   char *hostname_SE = NULL;
172   1   char wi_type;
173   1   boolean_ty submitCancelled = FALSE;
174   1   char **envVar;
175   1   int tmp;

177   1   /*
178   1    * The submitElement destructor should free these structures.
179   1    */
180   1   this_submit_files = malloc(sizeof(struct mark_summary));
181   1   total_submit_files = malloc(sizeof(struct mark_summary));

183   1   if((NULL == this_submit_files) ||
184   1      (NULL ==total_submit_files))
185   2   {
186   2     rec_api_log_csm(SUB_CSM_NOMEM, NULL);
187   2     rbe_log_stats(0, "Could not allocate memory RSTSL_Submit()");
```



```
188   2     return(EP_RB_RECOVER_NOMEM);
189   1   }

191   1   memset(this_submit_files, 0, sizeof(struct mark_summary));
192   1   memset(total_submit_files, 0, sizeof(struct mark_summary));



195   1   /*
197   1    * for Direct Connect, call its plugin version of Submit:
198   1    * Did not change plug in call for CLI arguments for Port and Name
199   1    */
200   1   if (rcp->rc_backup_app != 0)
201   1     return rcp-> currentPIptr-> piFuncArray[ PIFuncIndexSubmit ]
202   1                            ( rcp,
203   1                              hostName,
204   1                              policy,
205   1                              inPlace,
206   1                              directory,
207   1                              transport,
208   1                              submitObjID,
209   1                              progressCB );

211   1   *submitObjID = NewSubmitObject(&SOstatus);

213   1   if(E_SUCCESS != SOstatus)
214   2     {
215   2       rbe_log_stats(0, "Could not submit new object ID");
216   2       return (EP_RB_RECOVER_FATALERR);
217   1     }


220   1   if (!inPlace && ((hostName == NULL) || ('\0' == hostName[0])))
221   2     {
222   2       rbe_log_stats(0, "Hostname or inplace not set");
223   2       return(EP_RB_RECOVER_BAD_ARGS);
224   1     }

226   1   submitElemID = NewSubmitElement(*submitObjID, &SEstatus);

228   1   if(E_SUCCESS != SEstatus)
229   2     {
230   2       rbe_log_stats(0, "Could not create new submit elements");
231   2       return (EP_RB_RECOVER_FATALERR);
232   1     }


235   1   if(0 != SetSOBasics(*submitObjID,
236   1                       0,
237   1                       0,
238   1                       &SOstatus))

240   2     {
241   2       rbe_log_stats(0, "Could not set SO basics");
242   2       return (EP_RB_RECOVER_FATALERR);
243   1     }

245   1   /*
246   1    * Set up the environment variables
247   1    */
248   1   envVar = calloc(2,sizeof(char *));
249   1   envVar[0] = submitArgs->mapfile_env;
250   1   if (NULL != envVar[0])
251   2     {
252   2       if (0 == strcmp(envVar[0], "\0"))
253   3         {
```

```
254 3             envVar[0] = NULL;
255 2           }
256 1       }
257 1     envVar[1] = NULL;
258 1     if (NULL != envVar[0])
259 2       {
260 2         if (0 != SetSOExecutePhase(*submitObjID,NULL,NULL,envVar,&tmp))
261 3             {
262 3               rec_api_log_csm(FATAL_ERROR, NULL);
263 3               rbe_log_stats(
                        0, "Could not set the mapfile environment variable\n");
264 3               return(FATAL_ERROR);

266 2             }
267 1       }

269 1     /*
270 1      * Set the name of the client who initiated the call and the
271 1      * port to connect to.  Needs to be done before Direct connect call
272 1      */
273 1     if((0 != submitArgs->clientSocketPort) || (
                                        NULL != submitArgs->socketClientNm))
274 2       {
275 2         if ( 0 != SetSEEbciConnect(*submitObjID,
276 2                                    submitElemID,
277 2                                    submitArgs->socketClientNm,
278 2                                    submitArgs->clientSocketPort,
279 2                                    &SEstatus))
280 3             {
281 3               rbe_log_stats(
                              0, "Could not set socket port or client name");
282 3               return (EP_RB_RECOVER_FATALERR);
283 2             }
284 1       }

286 1       if(0 != SetSOAdminID(*submitObjID,
287 1                            (
            rcp->rc_recovery_flags & RC_RECFLAG_ADMINISTRATOR) ? 1 : 0,
288 1                            (
            rcp->rc_recovery_flags & RC_RECFLAG_SOURCE_SYSADMIN) ? 1 : 0,
289 1                            (
              rcp->rc_recovery_flags & RC_RECFLAG_DEST_SYSADMIN) ? 1 : 0,
290 1                           &SOstatus))
291 2       {
292 2         return (EP_RB_RECOVER_FATALERR);
293 1       }

295 1       if ((NULL != rcp->rc_human_uidname) &&
296 1            (NULL != rcp->rc_effective_uidname))
297 2       {
298 2         if(0 != SetSOUserID(*submitObjID,
299 2                             rcp->rc_human_uid,
300 2                             rcp->rc_human_uidname,
301 2                             rcp->rc_effective_uid,
302 2                             rcp->rc_effective_uidname,
303 2                             &SOstatus))
304 3           {
305 3             return (EP_RB_RECOVER_FATALERR);
306 2           }

308 1       }
309 1       else
310 2       {
311 2         rbe_log_stats(
             0, "Restore context has not set human_name and/or effective_name,
                                                          in RSTSL_Submit()");
```



```
312 2         return(EP_RB_RECOVER_BAD_CONTEXT);
313 1       }

315 1       if(0 != SetSOVMCheck(*submitObjID,
316 1                            (
                        rcp->rc_recovery_flags & RC_RECFLAG_NO_VM_CHECK),
317 1                            &SOstatus))

319 2       {
320 2         return (EP_RB_RECOVER_FATALERR);
321 1       }

323 1       /*
324 1        * Initialize this at the start of each recover within a single
325 1        * recovery session.
                              This is important if the destination directory
326 1        * gets changed from a specific location to simply "in place".
327 1        */

329 1       if (NULL != rcp->rc_client_dirtop)
330 2       {
331 2           free(rcp->rc_client_dirtop);
332 1       }

334 1       rcp->rc_client_dirtop = NULL;

336 1       if (rcp->rc_client_hostname)
337 2       {
338 2           free(rcp->rc_client_hostname);
339 1       }

341 2       {
342 2         /*
343 2          * We need to get information about this work item from
344 2          * from the config structure. Currently this is just the
345 2          * work item type.
346 2          */
347 2         RBC_WORKGROUP *wgp;
348 2         boolean_ty wi_found = FALSE;

350 2         if(NULL == rcp->rc_config)
351 3         {
352 3           rbe_log_stats(
          0, "Restore context has not set up the config is RSTSL_Submit()");
353 3           return(EP_RB_RECOVER_BAD_CONTEXT);
354 2         }

356 2         if(NULL == rcp->rc_workitem_name)
357 3         {
358 3           rbe_log_stats(
      0, "Restore context has not set up current work item in RSTSL_Submit(
                                                                        )");
359 3           return(EP_RB_RECOVER_BAD_CONTEXT);
360 2         }

362 2         wi_found = GetClientTypeFromConfig(rcp->rc_config,
363 2                                            rcp->rc_workitem_name,
364 2                                            &wi_type);
365 2         if (FALSE == wi_found)
366 3         {
367 3           rbe_log_stats(
                0, "Could not find the work item in the config structure in
                                                              RSTSL_Submit()");
368 3           return(EP_RB_RECOVER_BAD_CONTEXT);
369 2         }
370 1       }
```

```
371 1     if (inPlace)
372 2     {
373 2       if(NULL == rcp->rc_source_client_hostname)
374 3       {
375 3          return(EP_RB_RECOVER_BAD_CONTEXT);
376 2       }
377 2       else
378 3       {
379 3          hostname_SE = esl_strdup(rcp->rc_source_client_hostname);
380 2       }

382 1     }
383 1     else /* !inPlace */
384 2     {
385 2       /* hostName is checked above if !inPlace */

387 2       hostname_SE = (char *) hostName;
388 1     }
389 1     if (NULL == (rcp->rc_client_hostname = esl_strdup((
                                                    char *)hostname_SE)))
390 2     {
391 2          rec_api_log_csm(SUB_CSM_NOMEM, NULL);
392 2          rbe_log_stats(0, "Could not allocate memory RSTSL_Submit()");
393 2          return(EP_RB_RECOVER_NOMEM);
394 1     }

396 1     rcp->rc_overwrite_policy = policy;

398 1     /*
399 1      * fill_client_dirtop must always be called because
400 1      * cross recoveries for NOS clients require some special
401 1      * handling. The destination server name needs to be
402 1      * prepended along with a : in order for it to build the
403 1      * correct target string on the recover command sent to the client.
404 1      * fill_client_dirtop() handles this correctly.
405 1      */

407 1     if(0 != fill_client_dirtop2(rcp->rc_config,
408 1                                 inPlace,
409 1                                 (!inPlace) ? (char *) directory : (
                                                        char *) NULL,
410 1                                 rcp->rc_workitem_name,
411 1                                 hostname_SE,
412 1                                 &(rcp->rc_client_dirtop)))
413 2     {
414 2          rbe_log_stats(0, "Internal error:"\
415 2                        "Could not file client dirtop in RSTSL_Submit(
                                                                    )");

417 2          return(EP_RB_RECOVER_BAD_CONTEXT);
418 1     }

420 1     if (NULL == rcp->rc_client_dirtop)
421 2     {
422 2          rec_api_log_csm(SUB_CSM_NOMEM, NULL);
423 2          rbe_log_stats(0, "Could not allocate memory RSTSL_Submit()");
424 2          return(EP_RB_RECOVER_NOMEM);
425 1     }

427 1     if((NULL != rcp->rc_workitem_name) &&
428 1        (NULL != rcp->rc_template_name) &&
429 1        (NULL != rcp->rc_source_client_hostname))
430 2     {
431 2       if ( 0 != SetSEBasics(*submitObjID,
432 2                             submitElemID,
433 2                             rcp->rc_workitem_name,
```



```
434 2                             rcp->rc_template_name,
435 2                             rcp->rc_saveset_thread,
436 2                             rcp->rc_source_client_hostname,
437 2                             wi_type,
438 2                             &SEstatus))

440 3       {
441 3          return (EP_RB_RECOVER_FATALERR);
442 2       }
443 1     }
444 1     else
445 2     {
446 2       rbe_log_stats(
                          0, "Restore context has not set template_name,
              witem_name and/or source_client_name, in RSTSL_Submit()");
447 2       return(EP_RB_RECOVER_BAD_CONTEXT);
448 1     }



452 1     /* directory & hostName are check as input args */
453 1     /* What about transport ?? */

455 1     if(0 != SetSEDestination(*submitObjID,
456 1                              submitElemID,
457 1                              (char *)hostname_SE,
458 1                              inPlace,
459 1                              (char *)directory,
460 1                              policy,
461 1                              transport,
462 1                              &SEstatus))
463 2     {
464 2       return (EP_RB_RECOVER_FATALERR);
465 1     }

467 1     if(NULL != rcp->rc_client_dirtop)
468 2     {
469 2       if(0 != SetSEDirtop(*submitObjID,
470 2                           submitElemID,
471 2                           rcp->rc_client_dirtop,
472 2                           &SEstatus))
473 3       {
474 3          return (EP_RB_RECOVER_FATALERR);
475 2       }
476 1     }
477 1     else
478 2     {
479 2       rbe_log_stats(
              0, "Restore context has not set dirtop, in RSTSL_Submit()");
480 2       return(EP_RB_RECOVER_BAD_CONTEXT);
481 1     }

483 1     if((NULL != rcp->rc_client_scriptname) &&
484 1        (NULL != rcp->rc_client_rbuname))
485 2     {
486 2       if(0 != SetSEScriptName(*submitObjID,
487 2                               submitElemID,
488 2                               rcp->rc_client_scriptname,
489 2                               rcp->rc_client_rbuname,
490 2                               &SEstatus))
491 3       {
492 3          return (EP_RB_RECOVER_FATALERR);
493 2       }
494 1     }

496 1     else
```

```
497 2     {
498 2       rbe_log_stats(
             0, "Restore context has not set scriptname or client user_name,
                                                    in RSTSL_Submit()");
499 2       return(EP_RB_RECOVER_BAD_CONTEXT);
500 1     }

502 1     /*
503 1      * Progress callback intended to test for cancelation and
504 1      * to report progress on the submit to the user.
505 1      */
506 1     if(TRUE == progressCB(0))
507 2     {
508 2       /* Lets clean up here!
509 2        * right now there is no clean up routine for submitObjects.
510 2        */
511 2       rbe_log_stats(EP_RB_RECOVER_ABORT,"User abort during submit.");
512 2       submitCancelled = TRUE;
513 2       return(EP_RB_RECOVER_ABORT);
514 1     }
515 1     submit_fd = OpenSubmitFile(TRUE,
516 1                                *submitObjID,
517 1                                submitElemID,
518 1                                &SEstatus);


521 1     if(-1 == submit_fd)
522 2     {
523 2       return (EP_RB_RECOVER_FATALERR);
524 1     }

526 1     ret_status = push_submit_file(rcp,
527 1                                   submit_fd,
528 1                                   this_submit_files,
529 1                                   &this_submit_volumes,
530 1                                   total_submit_files,
531 1                                   &total_submit_volumes,
532 1                                   progressCB,
533 1                                   &submitCancelled);


536 1     CloseSubmitFile(submit_fd, TRUE, &SEstatus);

538 1     if(TRUE == submitCancelled)
539 2     {
540 2       /* Lets clean up here!
541 2        * right now there is no clean up routine for submitObjects.
542 2        */
543 2       rbe_log_stats(EP_RB_RECOVER_ABORT,"User abort during submit.");
544 2       return(EP_RB_RECOVER_ABORT);
545 1     }

547 1     if(-1 == ret_status)
548 2     {
549 2       return (EP_RB_RECOVER_FATALERR);
550 1     }

552 1     *ObjectsSubmitted = (unsigned int) ret_status;

554 1     if(0 != SetSOTotalSize(*submitObjID,
555 1                            total_submit_files,
556 1                            &SOstatus))
557 2     {
558 2       ; /* Not sure this one needs to be handled */
559 1     }
```



```
561 1     if(0 !=SetSOTotalVolumes(*submitObjID,
562 1                              total_submit_volumes,
563 1                              &SOstatus))
564 2     {
565 2       ; /* Not sure this one needs to be handled */
566 1     }

568 1     if(0 != SetSESummary(*submitObjID, submitElemID,
569 1                          this_submit_files, &SEstatus))
570 2     {
571 2       ; /* Not sure this one needs to be handled */
572 1     }
573 1     if(0 != SetSEVolumes(*submitObjID,
574 1                          submitElemID,
575 1                          this_submit_volumes,
576 1                          &SEstatus))
577 2     {
578 2       ; /* Not sure this one needs to be handled */
579 1     }

581 1     return( E_SUCCESS );
582   }
```

```
586     void
587     fill_client_dirtop(struct recover_context *rcx)
588   1 {
589   1     char buf[4096];
590   1     RBC_WORKITEM *pwi;
591   1     RBC_WORKGROUP *pwg;
592   1     boolean_ty cross_recover;


595   1     for (pwg = rcx->rc_config->pgrouplist; NULL != pwg;
                                                        pwg = pwg->next)
596   2     {
597   2          for (pwi = pwg->pwilist; NULL != pwi; pwi = pwi->next)
598   3          {
599   3              if (0 == strcmp(pwi->name, rcx->rc_workitem_name))
600   4              {
601   4                  goto gotit2;
602   3              }
603   2          }
604   1     }

606   1 gotit2:

608   1     /*
609   1      * if the dirtop already specifies a netware client
610   1      * target, remove it first.
611   1      */

613   1     if (rcx->rc_client_dirtop != NULL
614   1          && NULL != strchr(rcx->rc_client_dirtop, ':'))
615   2     {
616   2          return;
617   1     }

619   1     /*
620   1      * cross_recover is a boolean variable used to indicate a
                                                    cross-recover request.
621   1      * This will set the proper target for NOS clients and should NOT
                                                          affect others.
622   1      */

624   1     cross_recover = (NULL != rcx->rc_client_hostname) &&
625   1                (0 != strcmp(
                    rcx->rc_source_client_hostname, rcx->rc_client_hostname));

627   1     sprintf(buf, "%s%s%s",
628   1            ((NULL != pwi && NULL != pwi->nw_clnt_target)
629   1                ? (cross_recover
630   1                    ? rcx->rc_client_hostname
631   1                    : pwi->nw_clnt_target)
632   1                : ""),
633   1             ((NULL != pwi && NULL != pwi->nw_clnt_target) ? ":" : ""),
634   1             ((
          NULL != rcx->rc_client_dirtop) ? rcx->rc_client_dirtop : "/"));

636   1     if (rcx->rc_client_dirtop != NULL)
637   2     {
638   2          free (rcx->rc_client_dirtop);
639   1     }

641   1     rcx->rc_client_dirtop = esl_strdup(buf);
642   1     if (NULL == rcx->rc_client_dirtop)
643   2     {
```



```
644   2         rec_api_log_csm(SUB_CSM_NOMEM, NULL);
645   1     }
646     } /* fill_client_dirtop */
```

```
651   static int
652   push_submit_file(struct recover_context *rcx,
653                    int submit_fd,
654                    struct mark_summary *this_submit_files,
655                    ebvl_volidlist_ty **this_submit_volumes,
656                    struct mark_summary *total_submit_files,
657                    ebvl_volidlist_ty **total_submit_volumes,
658                    RSTSL_SubmitProgressProc progressCB,
659                    boolean_ty *submitCancelled)
660 1 {

662 1   int submit_cont = 1;
663 1   int retStatus = 0;
664 1   int bitfiles_pushed = 0;
665 1   ebfs_uid_ty prev_ebd;

667 1   if(NULL != submitCancelled)
668 1     *submitCancelled = FALSE;
669 1   else
670 1     return -1;

672 1   memset(&prev_ebd, 0, sizeof(ebfs_uid_ty));


675 1   if((NULL == rcx) ||
676 1      (NULL == this_submit_files) ||
677 1      (NULL == this_submit_volumes) ||
678 1      (NULL == total_submit_files) ||
679 1      (NULL == total_submit_volumes))
680 2   {
681 2     return -1;
682 1   }

684 1   if (submit_cont)
685 2   {

687 2           int plane;

689 2          /* MCAT_TOP is #define in mcat.h */
690 2           for (plane = (-rcx->rc_nplanes)+1; plane <= MCAT_TOP;
                                                                  plane++)
691 3           {
692 3               cat_descriptor *catd = mcat_getcatd(rcx->rc_mcp, plane);
693 3               char *marks = rcx->rc_marks[-plane];
694 3               long ntrlm;
695 3               long lmno;


698 3               if (catd == NULL)    /* should never happen */
699 4               {
700 4                   continue;
701 3               }
702 3               /* catdesc.h */
703 3               ntrlm = catd_ntrlm(catd);

705 3               for (lmno = 0; lmno < ntrlm; lmno++)
706 4               {
707 4                 /* RSLrbrmain.h */
708 4                   if (! TLMNO_MARKED(marks, lmno))
709 5                   {
710 5                       continue;
711 4                   }
```



```
713 4                   retStatus = push_bfinfo_to_submitfile(rcx,
714 4                                                         plane, catd,
                                                                 lmno,
715 4                                                         submit_fd,
716 4                                                         &prev_ebd,
717 4
                                                            this_submit_files,
718 4                                                      this_submit_volumes,
719 4
                                                             total_submit_files,
720 4                                                     total_submit_volumes);
721 4                   if (1 == retStatus)
722 5                   {
723 5                     bitfiles_pushed++;
724 4                   }
725 4                   if(bitfiles_pushed % 1024)
726 5                   {
727 5                     if(TRUE == progressCB(bitfiles_pushed))
728 6                     {

730 6                       rbe_log_stats(EP_RB_RECOVER_ABORT,
731 6                                     "User abort during submit.");
732 6                       *submitCancelled = TRUE;
733 6                       return(-1);
734 5                     }

736 4                   }
737 3               }
738 2           }

740 2           if(-1 == retStatus)
741 3           {
742 3             return -1;
743 2           }
744 2           return bitfiles_pushed;
745 1       }
746   }
```

```
749  /*
750   * Returns: -1 for file not submitted for restore, error encountered.
751   *           0 for file not submitted for restore,
                                                    NO error encountered.
752   *           1 for file subitted successfully.
753   */
754  static int
755  push_bfinfo_to_submitfile(struct recover_context *rcx,
756                            int plane,
757                            cat_descriptor *catd,
758                            long lmno,
759                            int fd,
760                            ebfs_uid_ty *prev_ebd,
761                            struct mark_summary *this_submit_files,
762                            ebvl_volidlist_ty **this_submit_volumes,
763                            struct mark_summary *total_submit_files,
764                            ebvl_volidlist_ty **total_submit_volumes)
765  1 {
766  1   /* Add mark support */
767  1     char              ebfsbfstr[34];
768  1     rbtree_elem_t     tlm;
769  1     rbcat_elem_t      clm;
770  1     long              catlmno;
771  1     cat_descriptor   *catlm_catd;
772  1     char              ebfsdirstr[34];
773  1     char              buf[100];
774  1     size_t            nbytes;
775  1     size_t            namesize = 0;
776  1     char             *fname = "<name unknown>";
777  1     char              *this_file;
778  1     eperrno            ep_status;
779  1     ebfs_uid_ty ebd;
780  1     ebfs_uid_ty zero_bitfileid;
781  1     char              *directives_list=NULL;
782  1     int               directives_size=0;


785  1     (void)catd_read_trlm(catd, lmno, &tlm, &this_file, (size_t *)NULL);

787  1     /*
788  1      * Get the corresponding catalog element.
789  1      * Note that it might come from a different
790  1      * plane if this tree element is a DS_NONE.
791  1      */

793  1     if (tlm.te_catelem != -1)
794  2     {
795  2          /*
796  2           * not DS_NONE -- the common case
797  2           */

799  2          catlmno = tlm.te_catelem;
800  2          catlm_catd = catd;
801  1     }
802  1     else
803  2     {
804  2          int clmplane;

806  2          /*
807  2           * This is a DS_NONE record.  Get
808  2           * the real corresponding catalog element.
809  2           */

811  2          dsnone_get_realcat(rcx, lmno, plane, &catlmno, &clmplane);
```



```
813  2          /*
814  2           * There is a special case for the root ("/")
815  2           * in the backup catalogs.  It's in the tree file
816  2           * but not in any catalog file.  This case can also
817  2           * occur for leading directories that are "above"
818  2           * the starting point for a work-item.  Silently
819  2           * ignore such directories, unless debugmode is on.
820  2           */

822  2          if (catlmno == -1)
823  3          {
824  3              size_t len;

826  3              (void)catd_read_trlm(catd, lmno, &tlm, &fname, &len);
827  3              if (len != 0 && debugmode) /* len 0 filters out "/" */
828  4              {
829  4                /*rbe_log_stats(
                   0, "** warning: cannot find catalog record for file %s;"
830  4                       " skipping it.", fname);*/
831  3              }
832  3              return 0;
833  2          }
834  2          catlm_catd = mcat_getcatd(rcx->rc_mcp, clmplane);
835  1      }
836  1      (void)catd_read_catlm(catlm_catd, catlmno, &clm, (char **)NULL);

838  1      add_to_summary(this_submit_files, &clm);
839  1      add_to_summary(total_submit_files, &clm);

841  1      *this_submit_volumes = ebvl_genvolidlist(*this_submit_volumes,
842  1                                              &clm.ce_bitfileID,
843  1                                              1,
                                                    ebvl_EbfsIdType_BitFile,
844  1                                              &ep_status);

846  1      if((NULL == *this_submit_volumes) || (0 != ep_status))
847  2      {
848  2        rbe_log_stats(
                  0, "** warning: cannot maintain volume list for submit."
849  2                       " skipping it.");
850  1      }

852  1      *total_submit_volumes = ebvl_genvolidlist(*total_submit_volumes,
853  1                                               &clm.ce_bitfileID,
854  1                                               1,
                                                     ebvl_EbfsIdType_BitFile,
855  1                                               &ep_status);

857  1      if((NULL == *total_submit_volumes) || (0 != ep_status))
858  2      {
859  2        rbe_log_stats(
                  0, "** warning: cannot maintain volume list for submit."
860  2                       " skipping it.");
861  1      }


864  1      memset(&zero_bitfileid, 0, sizeof(ebfs_uid_ty));


867  1      if(0 == memcmp(&zero_bitfileid,
868  1                     &clm.ce_bitfileID,
869  1                     sizeof(ebfs_uid_ty)))
870  2      {
871  2        (void)catd_read_catlm(catlm_catd, catlmno, &clm, &fname);
```

```
873 2       rbe_log_stats(
                    0, "** warning: There is no backup data to recover for "
874 2                  "file \"%s\"; skipping it.", fname);
875 2       return 0;
876 1     }



879 1     /*
880 1      * If this is a renamed element,
                                      must get the current name from cat
881 1      */



884 1     (void)catd_read_catlm(catlm_catd, catlmno, &clm, &fname);
885 1     if (clm.ce_status & CESTAT_RENAME)
886 2       {
887 2          /*
888 2           * name may contains substrings as in netware
889 2           */
890 2          namesize = (size_t)clm.ce_namelen;
891 1       }

893 1     if(0 != ssID2ebfd(&clm.ce_ssID, &ebd))
894 2     {
895 2       rbe_log_stats(
               0, "** warning: could not determine bitfile directory for "
896 2                  "file \"%s\"; skipping it.", fname);
897 2       return 0;
898 1     }

900 1     if ((NULL != rcx->recx_directives_p) || (NULL != fname))
901 2       {
902 2          directives_list=RSTSL_get_directives_for_file(
                                                 rcx->recx_directives_p,
903 2                                                       fname);
904 1       }
905 1     else
906 2       {
907 2          directives_list = NULL;
908 1       }
909 1     /* done in case string not null terminated */
910 1     if(directives_list !=NULL)
911 2       {
912 2          directives_size = strlen(directives_list);
913 1       }
914 1     else
915 2       {
916 2          directives_size = 0;
917 1       }

919 1     if(0 != WriteBitfileInfoToSubmitFile(fd,
920 1                                          &ebd,
921 1                                          &clm.ce_bitfileID,
922 1                                          clm.ce_mode,
923 1                                          namesize,
924 1                                          (
                                             namesize > 0) ? fname: NULL,
925 1                                          directives_size,
                                                    /* directive_size */
926 1                                          directives_list,
                                                       /* directives */
927 1                                          clm.ce_filesize,
928 1                                          prev_ebd))
929 2     {
930 2       rbe_log_stats(
                 0, "** warning: could not submit marked file for restore."
```



```
931 2                                " skipping file %s.", fname);
932 2       return 0;
933 1     }

935 1     memcpy(prev_ebd, &ebd, sizeof(ebfs_uid_ty));

937 1   #if 0     /* G.
                  Sachar: since buf is uninitialized and function is #if 0'd */
938 1     push_to_submitfile(fd, buf, strlen(buf));
939 1   #endif

941 1     return 1;

943 1     /*
944 1      * Creation and destruction of volid's now taks place
945 1      * outside of "go" command.
946 1      *
947 1      * now add bitfile id to volumes needed report
948 1      * if (rcx->ebvlok)
949 1      * {
950 1      *     rcx->ebvllist = ebvl_genvolidlist(
                                              rcx->ebvllist, &clm.ce_bitfileID,
951 1      *                                             1,
                                                ebvl_EbfsIdType_BitFile,
952 1      *                                            &ep_status);
953 1      * }
954 1      * if (rcx->ebvlok && NULL == rcx->ebvllist)
955 1      * {
956 1      *     fprintf(
                  stderr, "Unable to generate volumes needed report, %s (%d)",
957 1      *      e_get_error_text(ep_status), ep_status);
958 1      *     rcx->ebvlok = FALSE;
959 1      * }
960 1      */
961   }         /* end of push_bfinfo_to_submitfile() */
```

```
964     /* Returns: -1 for error encountered
965      *          otherwise number of bytes written
966      *
967      *
968      *
969      */
970
971     static int
972     push_to_submitfile(int fd,
973                        char *buf,
974                        uint_t nbytes)
975   1 {
976   1 #if 0
977   1     int wrote;
978   1     int save_errno;
979
980   1     if (debugmode)
981   2     {
982   2          (void)write(fileno(stdout), buf, nbytes);
983   1     }
984
985   1     wrote = looprw(fd, buf, (int)nbytes, write_no_eintr);
986   1     save_errno = errno;
987   1     if (wrote != (int)nbytes)
988   2     {
989   2       /*
990   2        * short write error
991   2        */
992   2       rbe_log_stats(0, "\n** Trouble writing submitfile.");
993   2       rbe_log_stats(
994                  0, "** wanted to write %d, wrote %d", nbytes, wrote);
995   1     }
996   1     errno = save_errno;
997   1     return wrote;
998   1 #endif
999   1     return nbytes;
1000  } /* end of push_to_submitfile() */
```



```
1002    /*
1003     * Convert an EBFS ID to string form.  Elide leading zeros.
1004     */
1005
1006    static void
1007    ebfsid2str_lz(ebfs_uid_ty *ebfsidp,
1008                  register char *buf)
1009  1 {
1010  1     register char *p;
1011  1     register char *q;
1012  1     unsigned long longvals[4];
1013  1     int i;
1014  1     char tmpbuf[ 33 ];
1015  1     char *hexdigits = "0123456789abcdef";
1016
1017
1018  1     memcpy(longvals, ebfsidp, 16);
1019
1020  1     q = tmpbuf;
1021
1022  1     for (i = 0; i < 4; i++)
1023  2     {
1024  2          int j;
1025  2          register unsigned long ul;
1026
1027  2          q += 8;
1028  2          ul = longvals[i];
1029
1030  2          for (j = 0; j < 8; j++)
1031  3          {
1032  3              *--q = hexdigits[ LONG2INT(ul & 0xF) ];
1033  3              ul >>= 4;
1034  2          }
1035  2          q += 8;
1036  1     }
1037
1038  1     tmpbuf[32] = '\0';
1039
1040  1     /*
1041  1      * Skip over leading 0 characters
1042  1      */
1043
1044  1     for (q = tmpbuf; *q == '0'; q++)
1045  2     {
1046  2          /* null */
1047  1     }
1048
1049  1     /*
1050  1      * Copy the rest, up to and including the comma, to output buf
1051  1      */
1052
1053  1     p = buf;
1054  1     while ((*p++ = *q++) != '\0')
1055  2     {
1056  2          /* null */
1057  1     }
1058  } /* end of ebfsid2str_lz() */
```

```
1061    static int
1062    ssID2ebfd(rbssID_t *ssidp,
1063              ebfs_uid_ty *ebfdp)
1064  1 {
1065  1     rbsaveset_t           ss;
1066  1     eperrno               err;


1069  1     /*
1070  1      * clobber it, to make failure to find match obvious
1071  1      */

1073  1    (void)memset((char *)ebfdp, 0, sizeof *ebfdp);


1076  1     if ((err = ss_find1(ssidp, &ss, 0)) == 0)
1077  2     {
1078  2       memcpy(ebfdp, &ss.ss_dirID, sizeof(ebfs_uid_ty));
1079  1     }

1081  1     return err ? -1 : 0;
1082    }       /* end of ssID2ebfd() */
```



```
1086    /*****************************************************************
                                                                  ********
1087    **
1088    ** Routine:  RSTSL_get_catalog_info
1089    **
1090    ** Inputs:  time      - The time of the backup that is being worked
                                                                      with
1091    **
1092    ** Outputs:  level    - reference to the level string to be output
1093    **                     will contain the level of the backup
1094    **           numrec    - reference to the string which will contain the
                                                                     number
1095    **                     of records for the backup
1096    **           catType   - reference to the string which will contain the
                                                                      type
1097    **                     of catalog for the specified backup
1098    **
1099    ** Purpose:  Function to retrieve backup level, catalog type,
                                                           and number of
1100    **           records and then retun it to the client
1101    **
1102    ** Return Codes: E_SUCCESS   - if the cattalog exists and able to
                                                                    return
1103    **                           information
1104    **                EP_RB_RECOVER_NO_CATALOG  - error getting the catalog
                                                                      info
1105    **
1106    *****************************************************************
                                                                  *******
1107    */
1108    eerrno_ty
1109    RSTSL_get_catalog_info( const time_t time,
1110                            char **level,
1111                            char **numrec,
1112                            char **catType)
1113        /* Called from re_get_catalog_info_1_svc RPC call*/
1114  1 {
1115  1       cat_descriptor *catdPtr;
                                    /* pointer to the catalog descriptor */
1116  1       rbcat_head_t cathdr;
                                /* pointer to head of a catalog retrieved from
1117  1                                  * the catalog descriptor
1118  1                                  */
1119  1       int plane_count=0;
                                /* used to keep track of the traversal through
1120  1                                  * the catalog plane structure
1121  1                                  */
1122  1       int number_of_planes;
                                   /* total number of planes to traverse */


1125  1       number_of_planes=-(rcp->rc_nplanes);
                                             /* must be negated because of
1126  1                                           * previous
                                                       implementation of
1127  1                                           * cat struct
1128  1                                           */

1130  1       do  /* traverse the list of planes in the catalog structure */
1131  2          {
1132  2            /* get a pointer to the plane I want staring at 0
1133  2             * and counting down to the number of planes in catalog
                                                                    struct
```

```
1134 2              */
1135 2          catdPtr = mcat_getcatd(rcp->rc_mcp, plane_count);
                                                    /* get the head */
1136 2          if (NULL == catdPtr)  /* if there are no catalogs */
1137 3            {
1138 3              return(EP_RB_RECOVER_NO_CATALOG);
1139 2            }

1141 2          if (0 != catd_get_cat_head(
                            catdPtr, &cathdr))  /* if there is no head
1142 2                                                 * there is
                                                       still not
1143 2                                                 *catalog
1144 2                                                 */
1145 3            {
1146 3              return(EP_RB_RECOVER_NO_CATALOG);
1147 2            }

1149 2          plane_count--;
                        /* decrement counter, must go into negatives */

1151 2          /* while the backup we are looking for has not been found,
                                                                    and
1152 2           * we have not traversed through the entire list
1153 2           */
1154 1          }while ((time !=cathdr.ch_time)&&(
                                      plane_count>=number_of_planes));

1156 1        *numrec=(char *)malloc( ULONG_TO_CHAR_SIZE );
                                                  /* must malloc memory
1157 1                                               * because cannot
                                                          strdup
1158 1                                               * a ulong
1159 1                                               */
1160 1        sprintf(*numrec,"%u",cathdr.ch_nelem);
                                               /* make the ulong a string
1161 1                                               * which is copied
                                                        from the
1162 1                                               *head of the
                                                          catalog
1163 1                                               */
1164 1        (*level) = (char *)calloc(1,2);
1165 1        (*level)[0] = cathdr.ch_level;

1167 1        switch (cathdr.ch_state)
1168 2           {
1169 2           case SSCAT_PARTIAL:
1170 2                *catType=esl_strdup("PARTIAL");
1171 2                break;
1172 2           case SSCAT_UNSORTED:
1173 2                *catType=esl_strdup("UNSORTED");
1174 2                break;
1175 2           case SSCAT_SORTED:
1176 2                *catType=esl_strdup("SORTED");
1177 2                break;
1178 2           case SSCAT_FULL:
1179 2                *catType=esl_strdup("FULL");
1180 2                break;
1181 2           case SSCAT_DELTA:
1182 2                *catType=esl_strdup("DELTA");
1183 2                break;
1184 2           case SSCAT_EXPIRED:
1185 2                *catType=esl_strdup("EXPIRED");
1186 2                break;
1187 2           case SSCAT_NONE:
1188 2                *catType=esl_strdup("NONE");
```



```
1189 2                break;
1190 2           default:
1191 2                *catType=esl_strdup("UNKNOWN");
1192 2                break;
1193 1           }

1195 1          return E_SUCCESS;
1196    }
```

```
1  /*****************************************************************************
                                                                          ******
2  **
3  ** File Name:   RSLstart.c
4  **
5  ** Copyright (c) 1998,1999 by EMC Corporation.
6  **
7  ** Purpose:
8  ** -----------
9  **       The intent of the contents of this file is to implement the
10 **       functions the control execution of the restore for the
                                                              Restore Service
11 **       Library.
12 **
13 **       These functions are provided to allow:
14 **      - creation of submit objects,
                                  which define the set of objects to be
15 **        restored and the scripts to be run before and after
                                                                 restoration,
16 **      - starting the restoral of a submit object.
17 **
18 ** The following functions comprise restoral management:
19 **
20 **      RSTSL_Start
21 **
22 **
23 ** Compile-Time Options:
24 **              This section must list any compile time definitions
25 **              which will affect this header.
26 **
27 *****************************************************************************
                                                                          *****/



30 /*
31  * Feature test switches.
32  * Standard defines required to turn on OS features go here.
33  *
34  * The following is required for code that uses POSIX API's.
35  * Remove for non-POSIX, non-portable code.
36  */

38 #define  _POSIX_SOURCE 1



41 /*
42  * System headers.
43  */

45 #include <sys/wait.h>

47 /*
48  * Epoch headers.
49  */
50 #include <eb/eb_port.h>
51 #include <eb/rb_log.h>
52 #include <ebutil/eb_normalize.h>
53 #include <ebutil/ebutil.h>
54 #include <ebreport/ebvl.h>



57 /*
58  * Local headers
59  */
```



```
62  #include <RSLinterns.h>
63  #include <RSLauxSupp.h>
64  #include <restore/EDMRESubmitApi.h>

66  #include <restore/REprogmsg.h>
67  #include <restore/dispatch_daemon.h>
68  #include <restore/EDMREProgressApi.h>

70  extern int RunExecutable(const boolean_ty ResetUid,
71                           const int RunUid,
72                           const char *starting_cwd,
73                           const char *executable_name,
74                           char **executable_argv,
75                           char **executable_env,
76                           int *run_exit_status,
77                           boolean_ty *run_cancelled,
78                           boolean_ty (*QuitTest)(void));

80  extern int RunWorkItemRestores(int, boolean_ty (*CancelRestoreTest)());

82  static eerrno_ty
83  ExecuteWorkItemRestore(int SubmitObjectID,
84                         boolean_ty (*QuitTest)(void));

86  static eerrno_ty
87  RunPrepareRestore(int SubmitObjectID,
88                    boolean_ty (*QuitTest)(void),
89                    int *PrepareExit);
90  static eerrno_ty
91  RunCleanupRestore(int SubmitObjectID,
92                    boolean_ty (*QuitTest)(void),
93                    int runphase_status,
94                    int *CleanupExit);

96  /*
97   * #defines, structures, typedefs local to this source file
98   */

100 #define STR_SURE(str) (str) ? str:""
101 #define REMOVE_NEWLINE(str)\
102 1 {\
103 1    int rem_nl_index;\
104 1    for(rem_nl_index = 0; str[rem_nl_index] != '\0'; rem_nl_index++)\
105 2      {\
106 2        if(str[rem_nl_index] == '\n')\
107 2           str[rem_nl_index] = '\0';\
108 1      }\
109 }

111 /*****************************************************************************
112  * Start
113  *
114  * This function begins execution of the restoral of the objects in a
115  * submit object.  Its progress and requests for operator input are
116  * returned via RSTSL_GetRestoreFeedback.
117  *
118  * Parameters:
119  *
120  * SubmitObjectID (
                  I) - ID of the submit object which describes the restore
121  * QuitTest       (I) - function to call to check for quit signal
122  *
123  *****************************************************************************/
124 eerrno_ty RSTSL_Start( int SubmitObjectID, boolean_ty (*QuitTest)(
                                                                      void))
125 1 {
```

```
127 1   int ret_pre;
128 1   int ret_exec;
129 1   int ret_post;
130 1   int ret_all_ok;
131 1   int PrepareExit = 0;
132 1   int CleanupExit = 0;
133 1   boolean_ty QuitFlag = FALSE;

135 1   char asctime[32];

137 1   memset(asctime, 0, 32);

139 1   (void)time(&rcp->rc_cmd_starttime);

141 1   (void)ctime_r(&rcp->rc_cmd_starttime, asctime, 32);

143 1   rcp->rc_cmd_last_waf_time = rcp->rc_cmd_starttime;

145 1   REMOVE_NEWLINE(asctime);

147 1   rbe_log_stats(0, "Restore Started at %s.", asctime);

149 1   rbe_log_stats(0, "Restore Started application type %s.",
150 1                 (rcp->rc_backup_app == 0)
151 1                         ? "Network"
152 1                         : ((struct pluginIDdata *)(
                                        rcp-> currentPIptr-> idData))
                                                              ->
153 1                                                        name );


155 1   rbe_log_stats(0, "Restore Started of "\
156 1                 "top level object: %s. template %s, trailset %s.",
157 1                 STR_SURE(rcp->rc_top_level_object_name),
158 1                 STR_SURE(rcp->rc_template_name),
159 1                 (rcp->rc_saveset_thread) ? "Alternate": "Primary");


162 1   rbe_log_stats(0, "Restore Started by user %s, Uid %d, Gid %d.",
163 1                 STR_SURE(rcp->rc_human_uidname),
164 1                 rcp->rc_human_uid, rcp->rc_human_gids[0]);

166 1   rbe_log_stats(0, "Restore Started with client destination %s.",
167 1                 STR_SURE(rcp->rc_client_hostname));

169 1   /* if not a network restore,
                        check if plugin has its own start function */


171 1   if ( rcp-> rc_backup_app != 0
172 1     && NULL != rcp-> currentPIptr-> piFuncArray[
                                        PIFuncIndexStartRestore ] )

173 2   {
174 2     setGlobalStatus( EDMRE_STATE_EXECUTE );
                                        /* set RE's internal status */
175 2     ret_exec = rcp-> currentPIptr-> piFuncArray[
                                          PIFuncIndexStartRestore ]
                                                (
176 2                                     rcp, SubmitObjectID, QuitTest);


178 2     if( QuitTest() )      /* check for abort before return */
179 3     {
180 3       rbe_log_stats( EP_RB_RECOVER_ABORT,
181 3                      "The restore was quit by the user during
                                                        execution.");
182 3       setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                        /* set RE's internal status */
```



```
183 3       return EP_RB_RECOVER_ABORT;
184 2     }

186 2     if (E_SUCCESS != ret_exec)
187 2       setGlobalStatus( EDMRE_STATE_FAILED );
                                        /* set RE's internal status */
188 2     else
189 2       setGlobalStatus( EDMRE_STATE_SUCCESSFUL );
                                        /* set RE's internal status */


191 2     return( ret_exec );
192 1   }

194 1   ret_pre = RunPrepareRestore(SubmitObjectID,
195 1                               QuitTest,
196 1                               &PrepareExit);

198 1   if(EP_RB_RECOVER_ABORT == ret_pre)
199 2   {
200 2     rbe_log_stats(EP_RB_RECOVER_ABORT,
201 2                   "The restore was quit by the user during
                                                        preparation.");
202 2     setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                        /* set RE's internal status */
203 2     return EP_RB_RECOVER_ABORT;
204 1   }
205 1   if(PrepareExit != 0)
206 2   {
207 2     rbe_log_stats(EP_RB_RECOVER_PREFAILED,
208 2                   "The restore failed during preparation. Exit %d",
209 2                   PrepareExit);

211 2     setGlobalStatus( EDMRE_STATE_FAILED );
                                        /* set RE's internal status */
212 2     return (EP_RB_RECOVER_PREFAILED);
213 1   }

215 1   setGlobalStatus( EDMRE_STATE_EXECUTE );
                                        /* set RE's internal status */

217 1   ret_exec = ExecuteWorkItemRestore(SubmitObjectID,
218 1                                     QuitTest);

220 1   QuitFlag = QuitTest();          /* check for abort before cleanup */


223 1   if (E_SUCCESS == (
                ret_all_ok = ret_exec) )      /* check if any WIs failed */
224 2   {
225 2     int local_stat;
226 2     EDMStats stats;
227 2     memset( &stats, 0, sizeof(EDMStats) );

229 2     if (0 != getRestoreStatus( 0, &stats, &local_stat ))
230 3     {
231 3       rbe_log_stats(
                  local_stat, "Internal error: Failed in getRestoreStatus.");
232 3       ret_exec = ret_all_ok = EP_RB_RECOVER_EXECUTEFAILED;
233 2     }
234 2     else if (stats.edm.failed)
235 3     {     /* if any workitems failed,
                                        its a failure for cleanup purposes */
236 3         if (0 == stats.edm.successful)
237 3           ret_all_ok = EP_RB_RECOVER_ALLFAIL;
238 3         else if (stats.edm.successful > stats.edm.failed)
239 3           ret_all_ok = EP_RB_RECOVER_FEWFAIL;
```

```
240 3           else
241 3             ret_all_ok = EP_RB_RECOVER_MANYFAIL;
242 2       }
243 1     }

245 1     ret_post = RunCleanupRestore(SubmitObjectID,
246 1                                  QuitTest,
247 1                                  ret_all_ok,
248 1                                  &CleanupExit);

250 1     if(QuitFlag)
251 2     {
252 2       rbe_log_stats(EP_RB_RECOVER_ABORT,
253 2                     "The restore was quit by the user during execution.
                                                                          ");
254 2       setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                          /* set RE's internal status */
255 2       return EP_RB_RECOVER_ABORT;
256 1     }

258 1     if (E_SUCCESS != ret_exec)      /* return execute status if it failed
                                                                          */
259 2     {
260 2       setGlobalStatus( EDMRE_STATE_FAILED );
                                          /* set RE's internal status */
261 2       return ret_exec;
262 1     }

264 1     if(EP_RB_RECOVER_ABORT == ret_post)
265 2     {
266 2       rbe_log_stats(EP_RB_RECOVER_ABORT,
267 2                     "The restore was quit by the user during cleanup.");
268 2       setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                          /* set RE's internal status */
269 2       return EP_RB_RECOVER_ABORT;
270 1     }

272 1     if( (CleanupExit != 0) || (E_SUCCESS != ret_post) )
273 2     {
274 2       rbe_log_stats(EP_RB_RECOVER_POSTFAILED,
275 2                     "The restore failed during cleanup. Exit %d",
276 2                     CleanupExit);

278 2       setGlobalStatus( EDMRE_STATE_FAILED );
                                          /* set RE's internal status */
279 2       return (EP_RB_RECOVER_POSTFAILED);

281 1     }

283 1     setGlobalStatus( EDMRE_STATE_SUCCESSFUL );
                                          /* set RE's internal status */

285 1     return( E_SUCCESS );

287   }         /* RSTSL_Start */
```



```
292   static eerrno_ty
293   ExecuteWorkItemRestore(int SubmitObjectID,
294                          boolean_ty (*QuitTest)(void))


297 1 {
298 1     int ret_RunWItem;
299 1     sm_push();

301 1     rcp->error_message[0] = 0;

303 1     if(0 != (ret_RunWItem = RunWorkItemRestores(
                                                SubmitObjectID, QuitTest)))
304 2     {
305 2       rbe_log_stats(0,"Internal error: Failed in RunWorkItemRestores");
306 1     }

308 1     sm_pop();

310 1     if (QuitTest() == TRUE)
311 1          return EP_RB_RECOVER_ABORT;

313 1     if (ret_RunWItem != 0)
314 1          return EP_RB_RECOVER_EXECUTEFAILED;

316 1     return E_SUCCESS;
317   }
```

```
320   #define EXECUTABLE_MAX 1024
321   static eerrno_ty
322   RunPrepareRestore(int SubmitObjectID,
323                     boolean_ty (*QuitTest)(void),
324                     int *PrepareExit)
325 1 {
326 1   char **prephaseargs = NULL;
327 1   char **prephaseenv = NULL;
328 1   int GetSOstatus = 0;
329 1   char preExecutable[EXECUTABLE_MAX];
330 1   boolean_ty restore_cancelled = FALSE;

332 1   *PrepareExit = 0;

334 1   /*
335 1    * GetSOPrePhase allocates prephaseargs & prephaseenv.
336 1    * This will need to be free'ed later.
337 1    *
338 1    */

340 1   if(0 != GetSOPrePhase(SubmitObjectID,
341 1                         preExecutable,
342 1                         EXECUTABLE_MAX,
343 1                         &prephaseargs,
344 1                         &prephaseenv,
345 1                         &GetSOstatus))
346 2   {
347 2     rbe_log_stats(0,"Internal error: Failed in GetSOPrePhase");
348 2     return (EP_RB_RECOVER_FATALERR);
349 1   }

351 1   if(0 != strcmp(preExecutable, ""))
352 2   {
353 2     setGlobalStatus( EDMRE_STATE_PREPHASE );
354 2     if(-1 == RunExecutable(FALSE,
355 2                            0,
356 2                            NULL,
357 2                            preExecutable,
358 2                            prephaseargs,
359 2                            prephaseenv,
360 2                            PrepareExit,
361 2                            &restore_cancelled,
362 2                            QuitTest))
363 3     {
364 3       rbe_log_stats(
              0,"Internal error: Failed in RunExecutable for prepare.");
365 3       return (EP_RB_RECOVER_FATALERR);
366 2     }
367 2     if(TRUE == restore_cancelled)
368 2       return (EP_RB_RECOVER_ABORT);

370 1   }

372 1   return( E_SUCCESS );
373   }
```



```
375   boolean_ty alwaysFalse() { return FALSE; }
```

```
377   static eerrno_ty
378   RunCleanupRestore(int SubmitObjectID,
379                     boolean_ty (*QuitTest)(void),
380                     int runphase_status,
381                     int *CleanupExit)
382 1 {
383
384 1   char **postphaseargs = NULL;
385 1   char **postphaseenv = NULL;
386 1   int GetSOstatus = 0;
387 1   char postExecutable[EXECUTABLE_MAX];
388 1   boolean_ty restore_cancelled = FALSE;
389 1   boolean_ty ignore_quit=FALSE;
390
391 1   *CleanupExit = 0;
392
393 1   /*
394 1    * GetSOPostPhase allocates postphaseargs & postphaseenv.
395 1    * This will need to be free'ed later.
396 1    *
397 1    */
398 1   PurgeTrailQueue();
399 1   if(0 != GetSOPostPhase(SubmitObjectID,
400 1                          postExecutable,
401 1                          EXECUTABLE_MAX,
402 1                          &postphaseargs,
403 1                          &postphaseenv,
404 1                          &GetSOstatus))
405 2   {
406 2     rbe_log_stats(0,"Internal error: Failed in GetSOPostPhase");
407 2     return (EP_RB_RECOVER_FATALERR);
408 1   }
409
410 1 #define RESTORE_BREAK       "RESTORE_BREAK="
411 1 #define RESTORE_BREAK_TRUE  "RESTORE_BREAK=T"
412 1 #define RESTORE_BREAK_ERROR "RESTORE_BREAK=E"
413
414 1   if(0 != strcmp(postExecutable, ""))
415 2    {
416 2      /*
417 2       * If a quit has been specified, we need to tweak the
418 2       * RESTORE_BREAK environment variable if set
419 2       */
420 2      char *abort=NULL;
421 2      if(QuitTest())
422 3        {
423 3           abort=RESTORE_BREAK_TRUE;
424 2        }
425 2      else if(0!=runphase_status)
426 3        {
427 3           abort=RESTORE_BREAK_ERROR;
428 2        }
429
430 2      /*
431 2       * ignore_quit is set to 1 when we have already processed a BREAK
432 2       * (CANCEL from gui), and are using the environment variable
433 2       * RESTORE_BREAK_E to signal the post restore script to clean
434 2       * up form this break.  When that happens, an ignore_quit value
435 2       * of 1 will cause actual quit signals to be ignores by the
436 2       * cleanup script, since we already know (via the environment
437 2       * variable) that we are in "cleanup mode" and no further signal
438 2       * intercpetion is necessary.
439 2       */
440 2      ignore_quit=FALSE;
441 2      if(NULL!=abort && NULL!=postphaseenv)
```



```
442 3        {
443 3           int isub=0;
444 3           char *cptr;
445 3           while(cptr=postphaseenv[isub])
446 4             {
447 4               if(strncmp(postphaseenv[isub],RESTORE_BREAK,strlen(
                                                    RESTORE_BREAK))==0)
448 5                 {
449 5                   postphaseenv[isub]=esl_strdup(abort);
450 5                   ignore_quit=TRUE;
451 5                   if(NULL==postphaseenv[isub])
452 6                     {
453 6                       rbe_log_stats(
          EP_RB_RECOVER_MALLOC_FAILURE,"Allocate failed in RSLstart.c");
454 6                       return EP_RB_RECOVER_POSTFAILED;
455 5                     }
456 4                 }
457 4               isub++;
458 3             }
459 2         }
460
461 2      setGlobalStatus( EDMRE_STATE_POSTPHASE );
                                              /* set RE's internal status */
462 2      if(-1 == RunExecutable(FALSE,
463 2                             0,
464 2                             NULL,
465 2                             postExecutable,
466 2                             postphaseargs,
467 2                             postphaseenv,
468 2                             CleanupExit,
469 2                             &restore_cancelled,
470 2                             ignore_quit?alwaysFalse:QuitTest))
471 3      {
472 3        rbe_log_stats(
               0,"Internal error: Failed in RunExecutable for cleanup.");
473 3        return (EP_RB_RECOVER_FATALERR);
474 2      }
475 2      if(TRUE == restore_cancelled)
476 2        return (EP_RB_RECOVER_ABORT);
477
478 1    }
479 1    return( E_SUCCESS );
480   }
```

```
482  static eerrno_ty
483  RunExecutionOverrideRestore(int SubmitObjectID,
484                              boolean_ty (*QuitTest)(void))
485  1 {
486
487  1   return( E_SUCCESS );
488  }
```



```
489  #undef EXECUTABLE_MAX
```